Two Sum

Given an array of integers, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

```
Input: numbers={2, 7, 11, 15}, target=9
Output: index1=1, index2=2
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Add Two Numbers

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

```
Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8
```

Solution 1

```java
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

Solution 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

Solution 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

From Leetcoder.

Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.

Solution 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

Solution 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

Solution 3

```
int lengthOfLongestSubstring(string s) {
        vector<int> dict(256, -1);
        int maxLen = 0, start = -1;
        for (int i = 0; i != s.length(); i++) {
            if (dict[s[i]] > start)
                start = dict[s[i]];
            dict[s[i]] = i;
            maxLen = max(maxLen, i - start);
        }
        return maxLen;
    }
```

written by lightmark original link here

From Leetcoder.

Median of Two Sorted Arrays

There are two sorted arrays **nums1** and **nums2** of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

Solution 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:

```python
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

Solution 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 observations:

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*Nth(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

Solution 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

From Leetcoder.

Longest Palindromic Substring

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

Solution 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

Solution 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

Solution 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

From Leetcoder.

ZigZag Conversion

The string `"PAYPALISHIRING"` is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P   A   H   N
A P L S I I G
Y   I   R
```

And then read line by line: `"PAHNAPLSIIGYIR"`
Write the code that will take a string and make this conversion given a number of rows:

```
string convert(string text, int nRows);
```

`convert("PAYPALISHIRING", 3)` should return `"PAHNAPLSIIGYIR"`.

Solution 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

Solution 2

```
/*n=numRows
Δ=2n−2     1                           2n−1                          4n−3
Δ=         2                     2n−2  2n                    4n−4    4n−2
Δ=         3                2n−3       2n+1               4n−5       .
Δ=         .            .               .                .            .
Δ=         .        n+2                 .            3n               .
Δ=         n−1  n+1                     3n−3    3n−1                  5n−5
Δ=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

Solution 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

Reverse digits of an integer.

```
Example1: x = 123, return 321
Example2: x = -123, return -321
```

click to show spoilers.

Have you thought about this?
Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!

If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.

Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?

For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.

Update (2014-11-10):
Test cases had been added to test the overflow behavior.

Solution 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

Solution 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

Solution 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

From Leetcoder.

String to Integer (atoi)

Implement `atoi` to convert a string to an integer.

Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

Update (2015-02-10):
The signature of the `C++` function had been updated. If you still see your function signature accepts a `const char *` argument, please click the reload button🔄 to reset your code definition.

spoilers alert... click to show requirements for atoi.

Requirements for atoi:
The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, INT_MAX (2147483647) or INT_MIN (-2147483648) is returned.

Solution 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

Solution 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

Solution 3

```
int myAtoi(string str) {
    int ret = 0, sign = 1, i = str.find_first_not_of(' '), base = INT_MAX / 10;
    if (str[i] == '+' || str[i] == '-') sign = str[i++] == '+' ?: -1;
    while (isdigit(str[i])) {
        if (ret > base || (ret == base && str[i] - '0' > 7))
            return sign > 0 ? INT_MAX : INT_MIN;
        ret = 10 * ret + (str[i++] - '0');
    }
    return sign * ret;
}
```

written by houzi original link here

From Leetcoder.

Palindrome Number

Determine whether an integer is a palindrome. Do this without extra space.

click to show spoilers.

Some hints:
Could negative integers be palindromes? (ie, -1)

If you are thinking of converting the integer to string, note the restriction of using extra space.

You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?

There is a more generic way of solving this problem.

Solution 1

```
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

Solution 2

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

Solution 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

From Leetcoder.

Regular Expression Matching

Implement regular expression matching with support for `'.'` and `'*'`.

```
'.' Matches any single character.
'*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be:
bool isMatch(const char *s, const char *p)

Some examples:
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "a*") → true
isMatch("aa", ".*") → true
isMatch("ab", ".*") → true
isMatch("aab", "c*a*b") → true
```

Solution 1

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

```
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

Solution 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

Solution 3

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

written by shawnForsythe original link here

From Leetcoder.

Container With Most Water

Given n non-negative integers $a_1, a_2, ..., a_n$, where each represents a point at coordinate (i, a_i). n vertical lines are drawn such that the two endpoints of line$i$ is at (i, a_i) and $(i, 0)$. Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Integer to Roman

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Roman to Integer

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

Solution 1

```
public class Solution {
    public int romanToInt(String s) {
        //: I (1) V (5) X (10) L (50) C (100) D (500) M (1000)
        // rules:位于大数的后面时就作为加数; 位于大数的前面就作为减数
        //eg: III=3,IV=4,VI=6,XIX=19,XX=20,XLV=45,MCMXXC=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

Solution 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

Solution 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

From Leetcoder.

Longest Common Prefix

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

Solution 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

Solution 2

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

```
    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

Solution 3

```
class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        string prefix = "";
        for(int idx=0; strs.size()>0; prefix+=strs[0][idx], idx++)
            for(int i=0; i<strs.size(); i++)
                if(idx >= strs[i].size() ||(i > 0 && strs[i][idx] != strs[i-1][id
x]))
                    return prefix;
        return prefix;
    }
};
```

written by huwenboshi original link here

From Leetcoder.

3Sum

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

Note:

- Elements in a triplet (a,b,c) must be in non-descending order. (ie, $a \le b \le c$)
- The solution set must not contain duplicate triplets.

```
For example, given array S = {-1 0 1 2 -1 -4},

A solution set is:
(-1, 0, 1)
(-1, -1, 2)
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

3Sum Closest

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

```
For example, given array S = {-1 2 1 -4}, and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
```

Solution 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

Solution 2

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.

```
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

Solution 3

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

From Leetcoder.

Letter Combinations of a Phone Number

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.

```
Input:Digit string "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
```

Note:
Although the above answer is in lexicographical order, your answer could be in any order you want.

Solution 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

Solution 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

Solution 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

From Leetcoder.

4Sum

Given an array S of n integers, are there elements a, b, c, and d in S such that $a + b + c + d$ = target? Find all unique quadruplets in the array which gives the sum of target.

Note:

- Elements in a quadruplet (a,b,c,d) must be in non-descending order. (ie, $a \le b \le c \le d$)
- The solution set must not contain duplicate quadruplets.

```
For example, given array S = {1 0 -1 0 -2 2}, and target = 0.

A solution set is:
(-1,  0, 0, 1)
(-2, -1, 1, 2)
(-2,  0, 0, 2)
```

Solution 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

Solution 2

```
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        vector<vector<int>> total;
        int n = nums.size();
        if(n<4)  return total;
        sort(nums.begin(),nums.end());
        for(int i=0;i<n-3;i++)
        {
            if(i>0&&nums[i]==nums[i-1]) continue;
            if(nums[i]+nums[i+1]+nums[i+2]+nums[i+3]>target) break;
            if(nums[i]+nums[n-3]+nums[n-2]+nums[n-1]<target) continue;
            for(int j=i+1;j<n-2;j++)
            {
                if(j>i+1&&nums[j]==nums[j-1]) continue;
                if(nums[i]+nums[j]+nums[j+1]+nums[j+2]>target) break;
                if(nums[i]+nums[j]+nums[n-2]+nums[n-1]<target) continue;
                int left=j+1,right=n-1;
                while(left<right){
                    int sum=nums[left]+nums[right]+nums[i]+nums[j];
                    if(sum<target) left++;
                    else if(sum>target) right--;
                    else{
                        total.push_back(vector<int>{nums[i],nums[j],nums[left],nu
ms[right]});
                        do{left++;}while(nums[left]==nums[left-1]&&left<right);
                        do{right--;}while(nums[right]==nums[right+1]&&left<right)
;
                    }
                }
            }
        }
        return total;
    }
};
```

written by cx1992 original link here

Solution 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

From Leetcoder.

Remove Nth Node From End of List

Given a linked list, remove the nth node from the end of list and return its head.

For example,

```
Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.
```

Note:
Given n will always be valid.
Try to do this in one pass.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Valid Parentheses

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

The brackets must close in the correct order, `"()"` and `"()[]{}"` are all valid but `"(]"` and `"([)]"` are not.

Solution 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

Solution 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

Solution 3

```
public class Solution {
    public boolean isValid(String s) {
        int length;

        do {
            length = s.length();
            s = s.replace("()", "").replace("{}", "").replace("[]", "");
        } while(length != s.length());

        return s.length() == 0;
    }
}
```

In this solution you essentially can remove parentheses that you know are valid until the string is empty. If the string is not empty, that means that the parentheses were malformed.

written by lough.r original link here

From Leetcoder.

Merge Two Sorted Lists

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

Solution 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

Solution 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

Solution 3

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

```
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

From Leetcoder.

Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given $n = 3$, a solution set is:

`"((()))", "(()())", "(())()", "()(())", "()()()"`

Solution 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

Solution 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

Solution 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

From Leetcoder.

Merge k Sorted Lists

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head.

For example,
Given `1->2->3->4`, you should return the list as `2->1->4->3`.

Your algorithm should use only constant space. You may **not** modify the values in the list, only nodes itself can be changed.

Solution 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

Solution 2

```
public ListNode swapPairs(ListNode head) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode current = dummy;
    while (current.next != null && current.next.next != null) {
        ListNode first = current.next;
        ListNode second = current.next.next;
        first.next = second.next;
        current.next = second;
        current.next.next = first;
        current = current.next.next;
    }
    return dummy.next;
}
```

written by tusizi original link here

Solution 3

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

From Leetcoder.

Reverse Nodes in k-Group

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

If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.

You may not alter the values in the nodes, only nodes itself may be changed.

Only constant memory is allowed.

For example,
Given this linked list: `1->2->3->4->5`

For k = 2, you should return: `2->1->4->3->5`

For k = 3, you should return: `3->2->1->4->5`

Solution 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

Solution 2

```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

Solution 3

```
-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

From Leetcoder.

Remove Duplicates from Sorted Array

Given a sorted array, remove the duplicates in place such that each element appear only *once* and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

For example,
Given input array *nums* = `[1,1,2]`,

Your function should return length = `2`, with the first two elements of *nums* being `1` and `2` respectively. It doesn't matter what you leave beyond the new length.

Solution 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

Solution 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

Solution 3

I don't understand why people insist on using old-style indexed looping. I much prefer the enhanced / range-based loops, they make things much cleaner.

C++

```
int removeDuplicates(vector<int>& nums) {
    int i = 0;
    for (int n : nums)
        if (!i || n > nums[i-1])
            nums[i++] = n;
    return i;
}
```

And to not need the `!i` check in the loop:

```
int removeDuplicates(vector<int>& nums) {
    int i = !nums.empty();
    for (int n : nums)
        if (n > nums[i-1])
            nums[i++] = n;
    return i;
}
```

Java

```
public int removeDuplicates(int[] nums) {
    int i = 0;
    for (int n : nums)
        if (i == 0 || n > nums[i-1])
            nums[i++] = n;
    return i;
}
```

And to not need the `i == 0` check in the loop:

```
public int removeDuplicates(int[] nums) {
    int i = nums.length > 0 ? 1 : 0;
    for (int n : nums)
        if (n > nums[i-1])
            nums[i++] = n;
    return i;
}
```

written by StefanPochmann original link here

Remove Element

Given an array and a value, remove all instances of that value in place and return the new length.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

Solution 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

Solution 2

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

Solution 3

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

From Leetcoder.

Implement strStr()

Implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Divide Two Integers

Divide two integers without using multiplication, division and mod operator.

If it is overflow, return MAX_INT.

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Substring with Concatenation of All Words

You are given a string, **s**, and a list of words, **words**, that are all of the same length. Find all starting indices of substring(s) in **s** that is a concatenation of each word in **words** exactly once and without any intervening characters.

For example, given:
s: `"barfoothefoobarman"`
words: `["foo", "bar"]`

You should return the indices: `[0,9]`.
(order does not matter).

Solution 1

```
// 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

Solution 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

Solution 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

From Leetcoder.

Next Permutation

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

`1,2,3` → `1,3,2`
`3,2,1` → `1,2,3`
`1,1,5` → `1,5,1`

Solution 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:

```java
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

Solution 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

Solution 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

From Leetcoder.

Longest Valid Parentheses

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring.

For `"(()"`, the longest valid parentheses substring is `"()"`, which has length = 2.

Another example is `")()())"`, where the longest valid parentheses substring is `"()()"`, which has length = 4.

Solution 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

Solution 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 ')'
    If s[i-1] is '(', longest[i] = longest[i-2] + 2
    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

Solution 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

From Leetcoder.

Search in Rotated Sorted Array

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

Solution 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

Solution 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

Solution 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.

```
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

From Leetcoder.

Search for a Range

Given a sorted array of integers, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of $O(\log n)$.

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

For example,
Given `[5, 7, 7, 8, 8, 10]` and target value 8,
return `[3, 4]`.

Solution 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

Solution 2

```
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

Solution 3

```
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

From Leetcoder.

Search Insert Position

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

You may assume no duplicates in the array.

Here are few examples.

`[1,3,5,6]`, $5 \rightarrow 2$

`[1,3,5,6]`, $2 \rightarrow 1$

`[1,3,5,6]`, $7 \rightarrow 4$

`[1,3,5,6]`, $0 \rightarrow 0$

Solution 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

Solution 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

Solution 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

From Leetcoder.

Valid Sudoku

Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules.

The Sudoku board could be partially filled, where empty cells are filled with the character `'.'`.

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

A partially filled sudoku which is valid.

Note:
A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character `'.'`.

You may assume that there will be only one unique solution.

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

A sudoku puzzle...

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

...and its solution numbers marked in red.

Solution 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

Solution 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];
```

```
        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

Solution 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 be tested 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);
```

```
    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

From Leetcoder.

Count and Say

The count-and-say sequence is the sequence of integers beginning as follows:
`1, 11, 21, 1211, 111221, ...`

`1` is read off as `"one 1"` or `11`.
`11` is read off as `"two 1s"` or `21`.
`21` is read off as `"one 2`, then `one 1"` or `1211`.

Given an integer n, generate the nth sequence.

Note: The sequence of integers will be represented as a string.

Solution 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

Solution 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

Solution 3

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

written by yuyibestman original link here

From Leetcoder.

Combination Sum

Given a set of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.

The **same** repeated number may be chosen from ***C*** unlimited number of times.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination (a_1, a_2, … , a_k) must be in non-descending order. (ie, $a_1 \le a_2 \le \ldots \le a_k$).
- The solution set must not contain duplicate combinations.

For example, given candidate set `2,3,6,7` and target `7`,
A solution set is:
`[7]`
`[2, 2, 3]`

Solution 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

Solution 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.

```
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

Solution 3

```
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

From Leetcoder.

Combination Sum II

Given a collection of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.

Each number in ***C*** may only be used **once** in the combination.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination (a_1, a_2, … , a_k) must be in non-descending order. (ie, $a_1 \le a_2 \le \ldots \le a_k$).
- The solution set must not contain duplicate combinations.

For example, given candidate set `10,1,2,7,6,1,5` and target `8`,
A solution set is:
`[1, 7]`
`[1, 2, 5]`
`[2, 6]`
`[1, 1, 6]`

Solution 1

```
 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

Solution 2

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

Solution 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

From Leetcoder.

First Missing Positive

Given an unsorted integer array, find the first missing positive integer.

For example,
Given `[1,2,0]` return `3`,
and `[3,4,-1,1]` return `2`.

Your algorithm should run in $O(n)$ time and uses constant space.

Solution 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

Solution 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.

```java
 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

Solution 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

From Leetcoder.

Trapping Rain Water

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,
Given `[0,1,0,2,1,0,1,3,2,1,2,1]`, return `6`.

The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Thanks Marcos for contributing this image!

Solution 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

Solution 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

Solution 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

From Leetcoder.

Multiply Strings

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note: The numbers can be arbitrarily large and are non-negative.

Solution 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

Solution 2

Remember how we do multiplication?

Start from right to left, perform multiplication on every pair of digits, and add them together. Let's draw the process! From the following draft, we can immediately conclude:

```
`num1[i] * num2[j]` will be placed at indices `[i + j`, `i + j + 1]`
```

Here is my solution. Hope it helps!

```java
public String multiply(String num1, String num2) {
    int m = num1.length(), n = num2.length();
    int[] pos = new int[m + n];

    for(int i = m - 1; i >= 0; i--) {
        for(int j = n - 1; j >= 0; j--) {
            int mul = (num1.charAt(i) - '0') * (num2.charAt(j) - '0');
            int p1 = i + j, p2 = i + j + 1;
            int sum = mul + pos[p2];

            pos[p1] += sum / 10;
            pos[p2] = (sum) % 10;
        }
    }

    StringBuilder sb = new StringBuilder();
    for(int p : pos) if(!(sb.length() == 0 && p == 0)) sb.append(p);
    return sb.length() == 0 ? "0" : sb.toString();
}
```

written by yavinci original link here

Solution 3

```
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.

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

written by lx223 original link here

From Leetcoder.

Wildcard Matching

Implement wildcard pattern matching with support for `'?'` and `'*'`.

```
'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

The function prototype should be:
bool isMatch(const char *s, const char *p)

Some examples:
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "*") → true
isMatch("aa", "a*") → true
isMatch("ab", "?*") → true
isMatch("aab", "c*a*b") → false
```

Solution 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

```
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

Solution 2

```python
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

```java
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

Solution 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`).

```
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

From Leetcoder.

Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example:
Given array A = `[2,3,1,1,4]`

The minimum number of jumps to reach the last index is `2`. (Jump `1` step from index 0 to 1, then `3` steps to the last index.)

Note:
You can assume that you can always reach the last index.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Permutations

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

For example,
`[1,2,3]` have the following permutations:
`[1,2,3]`, `[1,3,2]`, `[2,1,3]`, `[2,3,1]`, `[3,1,2]`, and `[3,2,1]`.

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,
`[1,1,2]` have the following unique permutations:
`[1,1,2]`, `[1,2,1]`, and `[2,1,1]`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Rotate Image

You are given an n x n 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up:
Could you do this in-place?

Solution 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

Solution 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

Solution 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

From Leetcoder.

Group Anagrams

Given an array of strings, group anagrams together.

For example, given: `["eat", "tea", "tan", "ate", "nat", "bat"]`,
Return:

```
[
  ["ate", "eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

Note:

1. For the return value, each *inner* list's elements must follow the lexicographic order.
2. All inputs will be in lower-case.

Solution 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

Solution 2

```
public class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        if (strs == null || strs.length == 0) return new ArrayList<List<String>>(
);
        Map<String, List<String>> map = new HashMap<String, List<String>>();
        Arrays.sort(strs);
        for (String s : strs) {
            char[] ca = s.toCharArray();
            Arrays.sort(ca);
            String keyStr = String.valueOf(ca);
            if (!map.containsKey(keyStr)) map.put(keyStr, new ArrayList<String>()
);
            map.get(keyStr).add(s);
        }
        return new ArrayList<List<String>>(map.values());
    }
}
```

written by wz366 original link here

Solution 3

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

From Leetcoder.

Pow(x, n)

Implement pow(x, n).

Solution 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

Solution 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

Solution 3

After reading some good sharing solutions, I'd like to show them together. You can see different ideas in the code.

1. nest myPow

```
double myPow(double x, int n) {
    if(n<0) return 1/x * myPow(1/x, -(n+1));
    if(n==0) return 1;
    if(n==2) return x*x;
    if(n%2==0) return myPow( myPow(x, n/2), 2);
    else return x*myPow( myPow(x, n/2), 2);
}
```

2. double myPow

```
double myPow(double x, int n) {
    if(n==0) return 1;
    double t = myPow(x,n/2);
    if(n%2) return n<0 ? 1/x*t*t : x*t*t;
    else return t*t;
}
```

3. double x

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

4. iterative one

```
double myPow(double x, int n) {
    if(n==0) return 1;
    if(n<0) {
        n = -n;
        x = 1/x;
    }
    double ans = 1;
    while(n>0){
        if(n&1) ans *= x;
        x *= x;
        n >>= 1;
    }
    return ans;
}
```

5. bit operation

see this solution

If you have other ideas, please leave it below. Thanks.

written by mingjun original link here

From Leetcoder.

N-Queens

The n-queens puzzle is the problem of placing n queens on an $n \times n$ chessboard such that no two queens attack each other.

One solution to the eight queens puzzle

Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

For example,
There exist two distinct solutions to the 4-queens puzzle:

```
[
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

Solution 1

In this problem, we can go row by row, and in each position, we need to check if the `column`, the `45° diagonal` and the `135° 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 45° 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 135° 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 45° diagonals is `2 * n - 1`, the number of 135° diagonals is also `2 * n - 1`. When reach `[row, col]`, the column No. is `col`, the 45° diagonal No. is `row + col` and the 135° 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 45° diagonals     5 135° 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 45° diagonal is `n + row + col` and the subscript of 135° 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 45° diagonal had a queen be
fore.
        flag[3 * n - 1] to flag[5 * n - 3] to indicate if the 135° 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

Solution 2

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

written by aqin original link here

Solution 3

ideas:
Use the `DFS` helper function to find solutions recursively. A solution will be found when the length of `queens` is equal to `n` (`queens` is a list of the indices of the queens).

In this problem, whenever a location `(x, y)` is occupied, any other locations `(p, q)` where `p + q == x + y` or `p - q == x - y` would be invalid. We can use this information to keep track of the indicators (`xy_dif` and `xy_sum`) of the invalid positions and then call DFS recursively with valid positions only.

At the end, we convert the result (a list of lists; each sublist is the indices of the queens) into the desire format.

```
def solveNQueens(self, n):
    def DFS(queens, xy_dif, xy_sum):
        p = len(queens)
        if p==n:
            result.append(queens)
            return None
        for q in range(n):
            if q not in queens and p-q not in xy_dif and p+q not in xy_sum:
                DFS(queens+[q], xy_dif+[p-q], xy_sum+[p+q])
    result = []
    DFS([],[],[])
    return [ ["."*i + "Q" + "."*(n-i-1) for i in sol] for sol in result]
```

written by cmc original link here

From Leetcoder.

N-Queens II

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.

One solution to the eight queens puzzle

Solution 1

```java
/**
 * 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

Solution 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

Solution 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

From Leetcoder.

Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array `[-2,1,-3,4,-1,2,1,-5,4]`,
the contiguous subarray `[4,-1,2,1]` has the largest sum = `6`.

click to show more practice.

More practice:
If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Spiral Matrix

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.

For example,
Given the following matrix:

```
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
```

You should return `[1,2,3,6,9,8,7,4,5]`.

Solution 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

Solution 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

Solution 3

```java
    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

From Leetcoder.

Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

For example:
A = `[2,3,1,1,4]`, return `true`.

A = `[3,2,1,0,4]`, return `false`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example,
Given `[1,3],[2,6],[8,10],[15,18]`,
return `[1,6],[8,10],[15,18]`.

Solution 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

Solution 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

Solution 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.

```
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

From Leetcoder.

Insert Interval

Given a set of *non-overlapping* intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Example 1:
Given intervals `[1,3],[6,9]`, insert and merge `[2,5]` in as `[1,5],[6,9]`.

Example 2:
Given `[1,2],[3,5],[6,7],[8,10],[12,16]`, insert and merge `[4,9]` in as `[1,2],[3,10],[12,16]`.

This is because the new interval `[4,9]` overlaps with `[3,5],[6,7],[8,10]`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Length of Last Word

Given a string *s* consists of upper/lower-case alphabets and empty space characters `' '`, return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example,
Given *s* = `"Hello World"`,
return `5`.

Solution 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

Solution 2

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

written by lvlolitte original link here

Solution 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

From Leetcoder.

Spiral Matrix II

Given an integer n, generate a square matrix filled with elements from 1 to n^2 in spiral order.

For example,
Given $n =$ `3`,

You should return the following matrix:

```
[
 [ 1, 2, 3 ],
 [ 8, 9, 4 ],
 [ 7, 6, 5 ]
]
```

Solution 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.

```python
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

Solution 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

Solution 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:

```
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

From Leetcoder.

Permutation Sequence

The set `[1,2,3,…,n]` contains a total of $n!$ unique permutations.

By listing and labeling all of the permutations in order,
We get the following sequence (ie, for $n = 3$):

1. `"123"`
2. `"132"`
3. `"213"`
4. `"231"`
5. `"312"`
6. `"321"`

Given n and k, return the kth permutation sequence.

Note: Given n will be between 1 and 9 inclusive.

Solution 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

Solution 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

Solution 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.

```
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

From Leetcoder.

Rotate List

Given a list, rotate the list to the right by k places, where k is non-negative.

For example:
Given `1->2->3->4->5->NULL` and k = `2`,
return `4->5->1->2->3->NULL`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Unique Paths

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

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

How many possible unique paths are there?

Above is a 3 x 7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

Solution 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

Solution 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

Solution 3

```
public class Solution {
    public int uniquePaths(int m, int n) {
        Integer[][] map = new Integer[m][n];
        for(int i = 0; i<m;i++){
            map[i][0] = 1;
        }
        for(int j= 0;j<n;j++){
            map[0][j]=1;
        }
        for(int i = 1;i<m;i++){
            for(int j = 1;j<n;j++){
                map[i][j] = map[i-1][j]+map[i][j-1];
            }
        }
        return map[m-1][n-1];
    }
}
```

The assumptions are

1. When (n==0||m==0) the function always returns 1 since the robot can't go left or up.
2. For all other cells. The result = uniquePaths(m-1,n)+uniquePaths(m,n-1)

Therefore I populated the edges with 1 first and use DP to get the full 2-D array.

Please give any suggestions on improving the code.

written by yx3110 original link here

From Leetcoder.

Unique Paths II

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as `1` and `0` respectively in the grid.

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is `2`.

Note: *m* and *n* will be at most 100.

Solution 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

Solution 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

Solution 3

Well, this problem is similar to **Unique Paths**. The introduction of obstacles only changes the boundary conditions and make some points unreachable (simply set to `0`).

Denote the number of paths to arrive at point `(i, j)` to be `P[i][j]`, the state equation is `P[i][j] = P[i - 1][j] + P[i][j - 1]` if `obstacleGrid[i][j] != 1` and `0` otherwise.

Now let's finish the boundary conditions. In the **Unique Paths** problem, we initialize `P[0][j] = 1, P[i][0] = 1` for all valid `i, j`. Now, due to obstacles, some boundary points are no longer reachable and need to be initialized to `0`. For example, if `obstacleGrid` is like `[0, 0, 1, 0, 0]`, then the last three points are not reachable and need to be initialized to be `0`. The result is `[1, 1, 0, 0, 0]`.

Now we can write down the following (unoptimized) code. Note that we pad the `obstacleGrid` by `1` and initialize `dp[0][1] = 1` to unify the boundary cases.

```
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];
    }
};
```

Well, the code is accepted but it has some obvious redundancy. There are two major concerns:

1. 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 unnecessary to maintain the full `m*n` matrix. Maintaining two columns is enough.
2. There are some cases that the loop can be terminated earlier. Suppose `obstacleGrid = [[0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0]]`, then we can see that it is impossible to reach the bottom-right corner after updating the second column since the number of paths to reach each element in the second column is `0`.

Taken these into considerations, we write down the following optimized code.

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        vector<int> pre(m, 0);
        vector<int> cur(m, 0);
        for (int i = 0; i < m; i++) {
            if (!obstacleGrid[i][0])
                pre[i] = 1;
            else break;
        }
        for (int j = 1; j < n; j++) {
            bool flag = false;
            if (!obstacleGrid[0][j]) {
                cur[0] = pre[0];
                if (cur[0]) flag = true;
            }
            else cur[0] = 0;
            for (int i = 1; i < m; i++) {
                if (!obstacleGrid[i][j]) {
                    cur[i] = cur[i - 1] + pre[i];
                    if (cur[i]) flag = true;
                }
                else cur[i] = 0;
            }
            if (!flag) return 0;
            swap(pre, cur);
        }
        return pre[m - 1];
    }
};
```

Further inspecting the above code, keeping two vectors only serve for the purpose of recovering `pre[i]`, which is simply `cur[i]` before its update. So we can use only one vector and the space is further optimized.

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        vector<int> cur(m, 0);
        for (int i = 0; i < m; i++) {
            if (!obstacleGrid[i][0])
                cur[i] = 1;
            else break;
        }
        for (int j = 1; j < n; j++) {
            bool flag = false;
            if (obstacleGrid[0][j])
                cur[0] = 0;
            else flag = true;
            for (int i = 1; i < m; i++) {
                if (!obstacleGrid[i][j]) {
                    cur[i] += cur[i - 1];
                    if (cur[i]) flag = true;
                }
                else cur[i] = 0;
            }
            if (!flag) return 0;
        }
        return cur[m - 1];
    }
};
```

written by jianchao.li.fighter original link here

From Leetcoder.

Minimum Path Sum

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which *minimizes* the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Solution 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

Solution 2

```
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

Solution 3

You can only reach a cell by going from its left or top neighbor.

```
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

From Leetcoder.

Valid Number

Validate if a given string is numeric.

Some examples:
`"0"` => `true`
`" 0.1 "` => `true`
`"abc"` => `false`
`"1 a"` => `false`
`"2e10"` => `true`

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

Update (2015-02-10):
The signature of the `C++` function had been updated. If you still see your function signature accepts a `const char *` argument, please click the reload button🔄 to reset your code definition.

Solution 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

Solution 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

Solution 3

All we need is to have a couple of flags so we can process the string in linear time:

```
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 ar `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

From Leetcoder.

Plus One

Given a non-negative number represented as an array of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Add Binary

Given two binary strings, return their sum (also a binary string).

For example,
a = `"11"`
b = `"1"`
Return `"100"`.

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Text Justification

Given an array of words and a length L, format the text such that each line has exactly L characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces `' '` when necessary so that each line has exactly L characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,
words: `["This", "is", "an", "example", "of", "text", "justification."]`
L: `16`.

Return the formatted lines as:

```
[
   "This    is    an",
   "example  of text",
   "justification.  "
]
```

Note: Each word is guaranteed not to exceed L in length.

click to show corner cases.

Corner Cases:

- A line other than the last line might contain only one word. What should you do in this case?
  In this case, that line should be left-justified.

Solution 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

Solution 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

Solution 3

```
class Solution {
public:
    vector<string> fullJustify(vector<string> &words, int L) {
        vector<string> ans;
        int begin = 0;
        while (begin < words.size()) {
            int last = begin;
            int linesize = words[begin++].size();
            while (begin < words.size() && linesize + 1 + words[begin].size() <=
L) {
                linesize += 1 + words[begin].size();
                begin++;
            }

            int spaces = 1, extra = 0;
            if (begin < words.size() && begin != last + 1) {
                spaces = (L - linesize) / (begin - last - 1) + 1;
                extra = (L - linesize) % (begin - last - 1);
            }

            ans.push_back(words[last++]);
            while (extra--) {
                ans.back().append(spaces+1, ' ');
                ans.back().append(words[last++]);
            }
            while (last < begin) {
                ans.back().append(spaces, ' ');
                ans.back().append(words[last++]);
            }
            ans.back().append(L-ans.back().size(), ' ');
        }

        return ans;
    }
};
```

written by xuewuxiao original link here

From Leetcoder.

Sqrt(x)

Implement `int sqrt(int x)`.

Compute and return the square root of x.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Climbing Stairs

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

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

Solution 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

Solution 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

Solution 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)

```ruby
def climb_stairs(n)
    a = b = 1
    n.times { a, b = b, a+b }
    a
end
```

C++ (0 ms)

```cpp
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)

```
int climbStairs(int n) {
    int a = 1, b = 1;
    while (n--)
        a = (b += a) - a;
    return a;
}
```

C# (48 ms)

```
public int ClimbStairs(int n) {
    int a = 1, b = 1;
    while (n-- > 0)
        a = (b += a) - a;
    return a;
}
```

Javascript (116 ms)

```
var climbStairs = function(n) {
    a = b = 1
    while (n--)
        a = (b += a) - a
    return a
};
```

written by StefanPochmann original link here

From Leetcoder.

Simplify Path

Given an absolute path for a file (Unix-style), simplify it.

For example,
path = `"/home/"`, => `"/home"`
path = `"/a/./b/../../c/"`, => `"/c"`

click to show corner cases.

Corner Cases:

- Did you consider the case where **path** = `"/../"`?
  In this case, you should return `"/"`.
- Another corner case is the path might contain multiple slashes `'/'` together, such as `"/home//foo/"`.
  In this case, you should ignore redundant slashes and return `"/home/foo"`.

Solution 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

Solution 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

Solution 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 1: double dot, pop dir
                 if(nameVect.size()>0)nameVect.pop_back();
            }else if(name==".."){//special case 2:singel dot, don`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

From Leetcoder.

Edit Distance

Given two words *word1* and *word2*, find the minimum number of steps required to convert *word1* to *word2*. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

a) Insert a character
b) Delete a character
c) Replace a character

Solution 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

Solution 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

Solution 3

http://www.stanford.edu/class/cs124/lec/med.pdf

written by vikram.kuruguntla original link here

From Leetcoder.

Set Matrix Zeroes

Given a m x n matrix, if an element is 0, set its entire row and column to 0. Do it in place.

click to show follow up.

Follow up:
Did you use extra space?
A straight forward solution using O(mn) space is probably a bad idea.
A simple improvement uses O($m + n$) space, but still not the best solution.
Could you devise a constant space solution?

Solution 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

Solution 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

Solution 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)

```cpp
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

From Leetcoder.

Search a 2D Matrix

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

For example,

Consider the following matrix:

```
[
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
```

Given **target** = `3`, return `true`.

Solution 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

Solution 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

Solution 3

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    // treat the matrix as an array, just taking care of indices
    // [0..n*m]
    // (row, col) -> row*n + col
    // i -> [i/n][i%n]
    if(matrix.empty() || matrix[0].empty())
    {
        return false;
    }
    int m = matrix.size(), n = matrix[0].size();
    int start = 0, end = m*n - 1;
    while(start <= end)
    {
        int mid = start + (end - start)/2;
        int e = matrix[mid/n][mid%n];
        if(target < e)
        {
            end = mid - 1;
        }
        else if(target > e)
        {
            start = mid + 1;
        }
        else
        {
            return true;
        }
    }
    return false;
}
```

written by xlingcc original link here

From Leetcoder.

Sort Colors

Given an array with *n* objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note:
You are not suppose to use the library's sort function for this problem.

click to show follow up.

Follow up:
A rather straight forward solution is a two-pass algorithm using counting sort.
First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

Solution 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

Solution 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

Solution 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

From Leetcoder.

Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

For example,
S = `"ADOBECODEBANC"`
T = `"ABC"`

Minimum window is `"BANC"`.

Note:
If there is no such window in S that covers all characters in T, return the empty string `""`.

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

Solution 1

I will first give the solution then show you the magic template.

The code of solving this problem is below. It might be the shortest among all solutions provided in Discuss.

```
string minWindow(string s, string t) {
        vector<int> map(128,0);
        for(auto c: t) map[c]++;
        int counter=t.size(), begin=0, end=0, d=INT_MAX, head=0;
        while(end<s.size()){
            if(map[s[end++]]-->0) counter--; //in t
            while(counter==0){ //valid
                if(end-begin<d)  d=end-(head=begin);
                if(map[s[begin++]]++==0) counter++;  //make it invalid
            }
        }
        return d==INT_MAX? "":s.substr(head, d);
    }
```

Here comes the template.

For most substring problem, we are given a string and need to find a substring of it which satisfy some restrictions. A general way is to use a hashmap assisted with two pointers. The template is given below.

```
int findSubstring(string s){
        vector<int> map(128,0);
        int counter; // check whether the substring is valid
        int begin=0, end=0; //two pointers, one point to tail and one  head
        int d; //the length of substring

        for() { /* initialize the hash map here */ }

        while(end<s.size()){

            if(map[s[end++]]-- ?){  /* modify counter here */ }

            while(/* counter condition */){

                 /* update d here if finding minimum*/

                //increase begin to make it invalid/valid again

                if(map[s[begin++]]++ ?){ /*modify counter here*/ }
            }

            /* update d here if finding maximum*/
        }
        return d;
  }
```

One thing needs to be mentioned is that when asked to find maximum substring,

we should update maximum after the inner while loop to guarantee that the substring is valid. On the other hand, when asked to find minimum substring, we should update minimum inside the inner while loop.

The code of solving **Longest Substring with At Most Two Distinct Characters** is below:

```
int lengthOfLongestSubstringTwoDistinct(string s) {
        vector<int> map(128, 0);
        int counter=0, begin=0, end=0, d=0;
        while(end<s.size()){
            if(map[s[end++]]++==0) counter++;
            while(counter>2) if(map[s[begin++]]--==1) counter--;
            d=max(d, end-begin);
        }
        return d;
    }
```

The code of solving **Longest Substring Without Repeating Characters** is below:

Update 01.04.2016, thanks @weiyi3 for advise.

```
int lengthOfLongestSubstring(string s) {
        vector<int> map(128,0);
        int counter=0, begin=0, end=0, d=0;
        while(end<s.size()){
            if(map[s[end++]]++>0) counter++;
            while(counter>0) if(map[s[begin++]]-->1) counter--;
            d=max(d, end-begin); //while valid, update d
        }
        return d;
    }
```

I think this post deserves some upvotes! :)

written by zjh08177 original link here

Solution 2

```
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

Solution 3

```
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

From Leetcoder.

Combinations

Given two integers n and k, return all possible combinations of k numbers out of $1 \ldots n$.

For example,
If $n = 4$ and $k = 2$, a solution is:

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

Solution 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

Solution 2

```
    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

Solution 3

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

From Leetcoder.

Subsets

Given a set of distinct integers, *nums*, return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example,
If ***nums*** = `[1,2,3]`, a solution is:

```
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example,
Given **board** =

```
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]
```

word = `"ABCCED"`, -> returns `true`,
word = `"SEE"`, -> returns `true`,
word = `"ABCB"`, -> returns `false`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates":
What if duplicates are allowed at most *twice*?

For example,
Given sorted array *nums* = `[1,1,1,2,2,3]`,

Your function should return length = `5`, with the first five elements of *nums* being `1`, `1`, `2`, `2` and `3`. It doesn't matter what you leave beyond the new length.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Search in Rotated Sorted Array II

Follow up for "Search in Rotated Sorted Array":
What if *duplicates* are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

Solution 1

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:

```java
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

Solution 2

```
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

Solution 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

From Leetcoder.

Remove Duplicates from Sorted List II

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.

For example,
Given `1->2->3->3->4->4->5`, return `1->2->5`.
Given `1->1->1->2->3`, return `2->3`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only *once*.

For example,
Given `1->1->2`, return `1->2`.
Given `1->1->2->3->3`, return `1->2->3`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Largest Rectangle in Histogram

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.

The largest rectangle is shown in the shaded area, which has area = `10` unit.

For example,
Given heights = `[2,1,5,6,2,3]`,
return `10`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

Solution 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

Solution 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.

```
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

Solution 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:

```
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

From Leetcoder.

Partition List

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,
Given `1->4->3->2->5->2` and $x = 3$,
return `1->2->2->4->3->5`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Scramble String

Given a string *s1*, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of *s1* = `"great"`:

```
    great
   /    \
  gr    eat
 / \    /  \
g   r  e   at
           / \
          a   t
```

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node `"gr"` and swap its two children, it produces a scrambled string `"rgeat"`.

```
    rgeat
   /    \
  rg    eat
 / \    /  \
r   g  e   at
           / \
          a   t
```

We say that `"rgeat"` is a scrambled string of `"great"`.

Similarly, if we continue to swap the children of nodes `"eat"` and `"at"`, it produces a scrambled string `"rgtae"`.

```
    rgtae
   /    \
  rg    tae
 / \    /  \
r   g  ta  e
       / \
      t   a
```

We say that `"rgtae"` is a scrambled string of `"great"`.

Given two strings *s1* and *s2* of the same length, determine if *s2* is a scrambled string of *s1*.

Solution 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

Solution 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

Solution 3

```
public class Solution {
    public boolean isScramble(String s1, String s2) {
        if (s1.equals(s2)) return true;

        int[] letters = new int[26];
        for (int i=0; i<s1.length(); i++) {
            letters[s1.charAt(i)-'a']++;
            letters[s2.charAt(i)-'a']--;
        }
        for (int i=0; i<26; i++) if (letters[i]!=0) return false;

        for (int i=1; i<s1.length(); i++) {
            if (isScramble(s1.substring(0,i), s2.substring(0,i))
             && isScramble(s1.substring(i), s2.substring(i))) return true;
            if (isScramble(s1.substring(0,i), s2.substring(s2.length()-i))
             && isScramble(s1.substring(i), s2.substring(0,s2.length()-i))) retur
n true;
        }
        return false;
    }
}
```

written by baifriend original link here

From Leetcoder.

Merge Sorted Array

Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.

Note:
You may assume that *nums1* has enough space (size that is greater or equal to $m + n$) to hold additional elements from *nums2*. The number of elements initialized in *nums1* and *nums2* are m and n respectively.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Gray Code

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given $n = 2$, return `[0,1,3,2]`. Its gray code sequence is:

```
00 - 0
01 - 1
11 - 3
10 - 2
```

Note:
For a given n, a gray code sequence is not uniquely defined.

For example, `[0,2,3,1]` is also a valid gray code sequence according to the above definition.

For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.

Solution 1

```
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

Solution 2

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

Solution 3

My idea is to generate the sequence iteratively. For example, when n=3, we can get the result based on n=2. 00,01,11,10 -> (000,001,011,010) (110,111,101,100). The middle two numbers only differ at their highest bit, while the rest numbers of part two are exactly symmetric of part one. It is easy to see its correctness. Code is simple:

```
public List<Integer> grayCode(int n) {
    List<Integer> rs=new ArrayList<Integer>();
    rs.add(0);
    for(int i=0;i<n;i++){
        int size=rs.size();
        for(int k=size-1;k>=0;k--)
            rs.add(rs.get(k) | 1<<i);
    }
    return rs;
}
```

written by yuyibestman original link here

From Leetcoder.

Subsets II

Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example,
If ***nums*** = `[1,2,2]`, a solution is:

```
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Decode Ways

A message containing letters from `A-Z` is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given an encoded message containing digits, determine the total number of ways to decode it.

For example,
Given encoded message `"12"`, it could be decoded as `"AB"` (1 2) or `"L"` (12).

The number of ways decoding `"12"` is 2.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Reverse Linked List II

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example:
Given `1->2->3->4->5->NULL`, $m = 2$ and $n = 4$,

return `1->4->3->2->5->NULL`.

Note:
Given m, n satisfy the following condition:
$1 \le m \le n \le$ length of list.

Solution 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 r
eversing
    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 w
ill 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

Solution 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

Solution 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

From Leetcoder.

Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:
Given `"25525511135"`,

return `["255.255.11.135", "255.255.111.35"]`. (Order does not matter)

Solution 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

Solution 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

Solution 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

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

For example:
Given binary tree `{1,#,2,3}`,

```
   1
    \
     2
    /
   3
```

return `[1,3,2]`.

Note: Recursive solution is trivial, could you do it iteratively?

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Unique Binary Search Trees II

Given n, generate all structurally unique **BST's** (binary search trees) that store values 1...n.

For example,
Given $n = 3$, your program should return all 5 unique BST's shown below.

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

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 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

Solution 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

Solution 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.

```
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

From Leetcoder.

Unique Binary Search Trees

Given n, how many structurally unique **BST's** (binary search trees) that store values 1...n?

For example,
Given $n = 3$, there are a total of 5 unique BST's.

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

Solution 1

The problem can be solved in a dynamic programming way. I'll explain the intuition and formulas in the following.

Given a sequence 1...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 1...(i-1) on its left side would lay on the left branch of the root, and similarly the right subsequence (i+1)...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 1...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) + … + 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) … G(n-1)`.

With the above explanation and formulas, here is the implementation in Java.

```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

Solution 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

Solution 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

From Leetcoder.

Interleaving String

Given *s1*, *s2*, *s3*, find whether *s3* is formed by the interleaving of *s1* and *s2*.

For example,
Given:
s1 = `"aabcc"`,
s2 = `"dbbca"`,

When *s3* = `"aadbbcbcac"`, return true.
When *s3* = `"aadbbbaccc"`, return false.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys**less than** the node's key.
- The right subtree of a node contains only nodes with keys**greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 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.
>
> "In-order Traversal": 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

Solution 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

Solution 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

From Leetcoder.

Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note:
A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?
confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 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:

```java
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

Solution 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

Solution 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.

```python
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

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

Solution 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

Solution 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

Solution 3

the idea is to use stack for preorder traverse

```
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

From Leetcoder.

Symmetric Tree

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

For example, this binary tree is symmetric:

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

But the following is not:

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

Note:
Bonus points if you could solve it both recursively and iteratively.

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 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

Solution 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

Solution 3

```
 public boolean isSymmetric(TreeNode root) {
    if(root==null) return true;
    return isMirror(root.left,root.right);
}
public boolean isMirror(TreeNode p, TreeNode q) {
    if(p==null && q==null) return true;
    if(p==null || q==null) return false;
    return (p.val==q.val) && isMirror(p.left,q.right) && isMirror(p.right,q.left);
}
```

written by yangneu2015 original link here

From Leetcoder.

Binary Tree Level Order Traversal

Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).

For example:
Given binary tree `{3,9,20,#,#,15,7}`,

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

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 1

```
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

Solution 2

```
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

Solution 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

From Leetcoder.

Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:
Given binary tree `{3,9,20,#,#,15,7}`,

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

return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 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

Solution 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

Solution 3

public class Solution { public List<List> zigzagLevelOrder(TreeNode root) { List<List> res = new ArrayList<>(); if(root == null) return res;

```
    Queue<TreeNode> q = new LinkedList<>();
    q.add(root);
    boolean order = true;
    int size = 1;

    while(!q.isEmpty()) {
        List<Integer> tmp = new ArrayList<>();
        for(int i = 0; i < size; ++i) {
            TreeNode n = q.poll();
            if(order) {
                tmp.add(n.val);
            } else {
                tmp.add(0, n.val);
            }
            if(n.left != null) q.add(n.left);
            if(n.right != null) q.add(n.right);
        }
        res.add(tmp);
        size = q.size();
        order = order ? false : true;
    }
    return res;
}
```

}

written by GraceLuLi original link here

From Leetcoder.

Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

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

Solution 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

Solution 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

Solution 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

From Leetcoder.

Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Binary Tree Level Order Traversal II

Given a binary tree, return the *bottom-up level order* traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example:
Given binary tree `{3,9,20,#,#,15,7}`,

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

return its bottom-up level order traversal as:

```
[
  [15,7],
  [9,20],
  [3]
]
```

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 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

Solution 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

Solution 3

The addFirst() method of LinkedLinked save us from reverse final result.

```
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

From Leetcoder.

Convert Sorted Array to Binary Search Tree

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

Solution 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

Solution 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

Solution 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

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Solution 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

Solution 2

```
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

Solution 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 )    // 寻找中间节点
        {
            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

From Leetcoder.

Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

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

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Solution 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

Solution 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

Solution 3

Why expected result for input of {1,2} is 2? Shouldn't it be 1?

written by ruilingrandy original link here

From Leetcoder.

Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example:
Given the below binary tree and `sum = 22`,

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \      \
        7    2      1
```

return true, as there exist a root-to-leaf path `5->4->11->2` which sum is 22.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Path Sum II

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:
Given the below binary tree and `sum = 22`,

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \    / \
        7    2  5   1
```

return

```
[
   [5,4,11,2],
   [5,8,4,5]
]
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

For example,
Given

```
          1
         / \
        2   5
       / \   \
      3   4   6
```

The flattened tree should look like:

```
   1
    \
     2
      \
       3
        \
         4
          \
           5
            \
             6
```

click to show hints.

Hints:
If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.

Solution 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

Solution 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

Solution 3

```
public void flatten(TreeNode root) {
        if (root == null) return;

        TreeNode left = root.left;
        TreeNode right = root.right;

        root.left = null;

        flatten(left);
        flatten(right);

        root.right = left;
        TreeNode cur = root;
        while (cur.right != null) cur = cur.right;
        cur.right = right;
    }
```

This solution is based on recursion. We simply flatten left and right subtree and paste each sublist to the right child of the root. (don't forget to set left child to null)

written by hanzhou87 original link here

From Leetcoder.

Distinct Subsequences

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"`

Return `3`.

Solution 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

Solution 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:

```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

Solution 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

From Leetcoder.

Populating Next Right Pointers in Each Node

Given a binary tree

```
struct TreeLinkNode {
  TreeLinkNode *left;
  TreeLinkNode *right;
  TreeLinkNode *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Note:

- You may only use constant extra space.
- You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,
Given the following perfect binary tree,

```
         1
       /  \
      2    3
     / \  / \
    4  5  6  7
```

After calling your function, the tree should look like:

```
         1 -> NULL
       /  \
      2 -> 3 -> NULL
     / \  / \
    4->5->6->7 -> NULL
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Populating Next Right Pointers in Each Node II

Follow up for problem "*Populating Next Right Pointers in Each Node*".

What if the given tree could be any binary tree? Would your previous solution still work?

Note:

- You may only use constant extra space.

For example,
Given the following binary tree,

```
         1
       /  \
      2    3
     / \    \
    4   5    7
```

After calling your function, the tree should look like:

```
         1 -> NULL
       /  \
      2 -> 3 -> NULL
     / \    \
    4-> 5 -> 7 -> NULL
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Pascal's Triangle

Given *numRows*, generate the first *numRows* of Pascal's triangle.

For example, given *numRows* = 5,
Return

```
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Pascal's Triangle II

Given an index k, return the kth row of the Pascal's triangle.

For example, given $k = 3$,
Return `[1,3,3,1]`.

Note:
Could you optimize your algorithm to use only $O(k)$ extra space?

Solution 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

Solution 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

Solution 3

```
public List<Integer> getRow(int rowIndex) {
    List<Integer> res = new ArrayList<Integer>();
    for(int i = 0;i<rowIndex+1;i++) {
            res.add(1);
            for(int j=i-1;j>0;j--) {
                res.set(j, res.get(j-1)+res.get(j));
            }
    }
    return res;
}
```

written by ggcrosemond original link here

From Leetcoder.

Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

The minimum path sum from top to bottom is `11` (i.e., 2 + 3 + 5 + 1 = 11).

Note:
Bonus point if you are able to do this using only $O(n)$ extra space, where n is the total number of rows in the triangle.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Best Time to Buy and Sell Stock

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Best Time to Buy and Sell Stock II

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution 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

Solution 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

Solution 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

From Leetcoder.

Best Time to Buy and Sell Stock III

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most *two* transactions.

Note:
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution 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

Solution 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

Solution 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

From Leetcoder.

Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.

For example:
Given the below binary tree,

```
      1
     / \
    2   3
```

Return `6`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,
`"A man, a plan, a canal: Panama"` is a palindrome.
`"race a car"` is *not* a palindrome.

Note:
Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

Solution 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

Solution 2

```
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

Solution 3

```
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

From Leetcoder.

Word Ladder II

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the word list

For example,

Given:
beginWord = `"hit"`
endWord = `"cog"`
wordList = `["hot","dot","dog","lot","log"]`

Return

```
[
  ["hit","hot","dot","dog","cog"],
  ["hit","hot","lot","log","cog"]
]
```

Note:

- All words have the same length.
- All words contain only lowercase alphabetic characters.

Solution 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;
                    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

Solution 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);
```

```
        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

Solution 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

From Leetcoder.

Word Ladder

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the word list

For example,

Given:
beginWord = `"hit"`
endWord = `"cog"`
wordList = `["hot","dot","dog","lot","log"]`

As one shortest transformation is `"hit" -> "hot" -> "dot" -> "dog" -> "cog"`,
return its length `5`.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.

Solution 1

```
//BFS,  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

Solution 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

Solution 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

From Leetcoder.

Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example,
Given `[100, 4, 200, 1, 3, 2]`,
The longest consecutive elements sequence is `[1, 2, 3, 4]`. Return its length: `4`.

Your algorithm should run in O(n) complexity.

Solution 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 dchen0215 original link here

Solution 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

Solution 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

From Leetcoder.

Sum Root to Leaf Numbers

Given a binary tree containing digits from `0-9` only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path `1->2->3` which represents the number `123`.

Find the total sum of all root-to-leaf numbers.

For example,

```
    1
   / \
  2   3
```

The root-to-leaf path `1->2` represents the number `12`.
The root-to-leaf path `1->3` represents the number `13`.

Return the sum = 12 + 13 = `25`.

Solution 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

Solution 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

Solution 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 node'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

From Leetcoder.

Surrounded Regions

Given a 2D board containing `'X'` and `'O'`, capture all regions surrounded by `'X'`.

A region is captured by flipping all `'O'`s into `'X'`s in that surrounded region.

For example,

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

Solution 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

Solution 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) {
        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

Solution 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

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

Return all possible palindrome partitioning of *s*.

For example, given *s* = `"aab"`,
Return

```
[
  ["aa","b"],
  ["a","a","b"]
]
```

Solution 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

Solution 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()

```java
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

Solution 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

From Leetcoder.

Palindrome Partitioning II

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

Return the minimum cuts needed for a palindrome partitioning of *s*.

For example, given *s* = `"aab"`,
Return `1` since the palindrome partitioning `["aa","b"]` could be produced using 1 cut.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Clone Graph

Clone an undirected graph. Each node in the graph contains a `label` and a list of its `neighbors`.

OJ's undirected graph serialization:
Nodes are labeled uniquely.

We use `#` as a separator for each node, and `,` as a separator for node label and each neighbor of the node.
As an example, consider the serialized graph `{0,1,2#1,2#2,2}`.

The graph has a total of three nodes, and therefore contains three parts as separated by `#`.

1. First node is labeled as `0`. Connect node `0` to both nodes `1` and `2`.
2. Second node is labeled as `1`. Connect node `1` to node `2`.
3. Third node is labeled as `2`. Connect node `2` to node `2` (itself), thus forming a self-cycle.

Visually, the graph looks like the following:

```
       1
      / \
     /   \
    0 --- 2
         / \
         \_/
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Gas Station

There are N gas stations along a circular route, where the amount of gas at station i is `gas[i]`.

You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

Note:
The solution is guaranteed to be unique.

Solution 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

Solution 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

Solution 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

There are N children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

Solution 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

Solution 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.

```
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

Solution 3

```
public int candy(int[] ratings) {
    int candies[] = new int[ratings.length];
    Arrays.fill(candies, 1);// Give each child 1 candy

    for (int i = 1; i < candies.length; i++){// Scan from left to right, to make
sure right higher rated child gets 1 more candy than left lower rated child
        if (ratings[i] > ratings[i - 1]) candies[i] = (candies[i - 1] + 1);
    }

    for (int i = candies.length - 2; i >= 0; i--) {// Scan from right to left, to
make sure left higher rated child gets 1 more candy than right lower rated child
        if (ratings[i] > ratings[i + 1]) candies[i] = Math.max(candies[i], (candi
es[i + 1] + 1));
    }

    int sum = 0;
    for (int candy : candies)
        sum += candy;
    return sum;
}
```

written by Pixel_ original link here

From Leetcoder.

Single Number

Given an array of integers, every element appears *twice* except for one. Find that single one.

Note:
Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Solution 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

Solution 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

Solution 3

we use bitwise XOR to solve this problem :

first , we have to know the bitwise XOR in java

1. **0 ^ N = N**
2. **N ^ N = 0**

So..... if N is the single number

N1 ^ N1 ^ N2 ^ N2 ^.....^ Nx ^ Nx ^ N

= (N1^N1) ^ (N2^N2) ^.....^ (Nx^Nx) ^ N

= 0 ^ 0 ^ .....^ 0 ^ N

= N

```
public int singleNumber(int[] nums) {
    int ans =0;

    int len = nums.length;
    for(int i=0;i!=len;i++)
        ans ^= nums[i];

    return ans;

}
```

written by Nkeys original link here

From Leetcoder.

Single Number II

Given an array of integers, every element appears *three* times except for one. Find that single one.

Note:
Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Solution 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 against1 original link here

Solution 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

Solution 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

From Leetcoder.

Copy List with Random Pointer

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

Solution 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:

```java
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

Solution 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:

```java
/**
 * 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

Solution 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

From Leetcoder.

Word Break

Given a string *s* and a dictionary of words *dict*, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words.

For example, given
s = `"leetcode"`,
dict = `["leet", "code"]`.

Return true because `"leetcode"` can be segmented as `"leet code"`.

Solution 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

Solution 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

Solution 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

Given a string *s* and a dictionary of words *dict*, add spaces in *s* to construct a sentence where each word is a valid dictionary word.

Return all such possible sentences.

For example, given
s = `"catsanddog"`,
dict = `["cat", "cats", "and", "sand", "dog"]`.

A solution is `["cats and dog", "cat sand dog"]`.

Solution 1

```
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

Solution 2

```
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

Solution 3

firstly I used DP from head of the string to traverse the dp-map: and then got a “Time Limit Exceeded” 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 “Time Limit Exceeded”

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

From Leetcoder.

Linked List Cycle

Given a linked list, determine if it has a cycle in it.

Follow up:
Can you solve it without using extra space?

Solution 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

Solution 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

Solution 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

From Leetcoder.

Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.

Note: Do not modify the linked list.

Follow up:
Can you solve it without using extra space?

Solution 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

Solution 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

Solution 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

From Leetcoder.

Reorder List

Given a singly linked list L: $L_0 \rightarrow L_1 \rightarrow \ldots \rightarrow L_{n-1} \rightarrow L_n$,
reorder it to: $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \ldots$

You must do this in-place without altering the nodes' values.

For example,
Given `{1,2,3,4}`, reorder it to `{1,4,2,3}`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Binary Tree Preorder Traversal

Given a binary tree, return the *preorder* traversal of its nodes' values.

For example:
Given binary tree `{1,#,2,3}`,

```
   1
    \
     2
    /
   3
```

return `[1,2,3]`.

Note: Recursive solution is trivial, could you do it iteratively?

Solution 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

Solution 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

Solution 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

From Leetcoder.

Binary Tree Postorder Traversal

Given a binary tree, return the *postorder* traversal of its nodes' values.

For example:
Given binary tree `{1,#,2,3}`,

```
   1
    \
     2
    /
   3
```

return `[3,2,1]`.

Note: Recursive solution is trivial, could you do it iteratively?

Solution 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

Solution 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

Solution 3

Here I summarize the iterative implementation for preorder, inorder, and postorder traverse.

Pre Order Traverse

```
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    Deque<TreeNode> stack = new ArrayDeque<>();
    TreeNode p = root;
    while(!stack.isEmpty() || p != null) {
        if(p != null) {
            stack.push(p);
            result.add(p.val);  // Add before going to children
            p = p.left;
        } else {
            TreeNode node = stack.pop();
            p = node.right;
        }
    }
    return result;
}
```

In Order Traverse

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    Deque<TreeNode> stack = new ArrayDeque<>();
    TreeNode p = root;
    while(!stack.isEmpty() || p != null) {
        if(p != null) {
            stack.push(p);
            p = p.left;
        } else {
            TreeNode node = stack.pop();
            result.add(node.val);  // Add after all left children
            p = node.right;
        }
    }
    return result;
}
```

Post Order Traverse

```java
public List<Integer> postorderTraversal(TreeNode root) {
    LinkedList<Integer> result = new LinkedList<>();
    Deque<TreeNode> stack = new ArrayDeque<>();
    TreeNode p = root;
    while(!stack.isEmpty() || p != null) {
        if(p != null) {
            stack.push(p);
            result.addFirst(p.val);  // Reverse the process of preorder
            p = p.right;             // Reverse the process of preorder
        } else {
            TreeNode node = stack.pop();
            p = node.left;           // Reverse the process of preorder
        }
    }
    return result;
}
```

written by yavinci original link here

From Leetcoder.

LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: `get` and `set`.

`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
`set(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

Solution 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;
```

```
    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

Solution 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

Solution 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

Sort a linked list using insertion sort.

Solution 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

Solution 2

One of the quotes is

> For God's sake, don't try sorting a linked list during the interview

http://steve-yegge.blogspot.nl/2008/03/get-that-job-at-google.html
So it might be better to actually copy the values into an array and sort them there.
written by orekhov.volodya original link here

Solution 3

```
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

From Leetcoder.

Sort List

Sort a linked list in $O(n \log n)$ time using constant space complexity.

Solution 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

Solution 2

```java
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

Solution 3

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

```
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

From Leetcoder.

Max Points on a Line

Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are `+`, `-`, `*`, `/`. Each operand may be an integer or another expression.

Some examples:

```
["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Reverse Words in a String

Given an input string, reverse the string word by word.

For example,
Given s = "`the sky is blue`",
return "`blue is sky the`".

Update (2015-02-12):
For C programmers: Try to solve it *in-place* in $O(1)$ space.

click to show clarification.

Clarification:

- What constitutes a word?
  A sequence of non-space characters constitutes a word.
- Could the input string contain leading or trailing spaces?
  Yes. However, your reversed string should not contain leading or trailing spaces.
- How about multiple spaces between two words?
  Reduce them to a single space in the reversed string.

Solution 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

Solution 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

Solution 3

```
String[] parts = s.trim().split("\\s+");
String out = "";
if (parts.length > 0) {
    for (int i = parts.length - 1; i > 0; i--) {
        out += parts[i] + " ";
    }
    out += parts[0];
}
return out;
```

I'm splitting on the regex for one-or-more whitespace, this takes care of multiple spaces/tabs/newlines/etc in the input. Since the input could have leading/trailing whitespace, which would result in empty matches, I first trim the input string.

Now there could be three possibilities: 1. The input is empty, return an empty string directly. 2. The input contains only one part, return that part. 3. The input contains multiple parts, reverse the order, making sure we don't end with a trailing space.

Obviously this is not the fastest or most memory efficient way to solve the problem, but optimizations should *only* be done when they are needed. Readable code is usually more important than efficient code.

How to make it efficient?

1. Use a StringBuilder to concatenate the string parts, instead of concatenating strings directly. This will (I assume) build something like a linked-list of string parts, and only allocate the new string when you need it, instead of on each concatenation.
2. Iterate over the string, instead of using trim/split. Store the index of the last character in the word, when you find the first character, copy the substring to the output string.
3. Instead of using substring, insert the word-characters directly in the StringBuilder. Assuming they're using a linked-list or tree, this could be a whole last faster.

written by daniel12 original link here

From Leetcoder.

Maximum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array `[2,3,-2,4]`,
the contiguous subarray `[2,3]` has the largest product = `6`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Find Minimum in Rotated Sorted Array

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

Find the minimum element.

You may assume no duplicate exists in the array.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Find Minimum in Rotated Sorted Array II

> *Follow up* for "Find Minimum in Rotated Sorted Array":
> What if *duplicates* are allowed?
>
> Would this affect the run-time complexity? How and why?

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

Find the minimum element.

The array may contain duplicates.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Min Stack

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

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

Solution 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

Solution 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

Solution 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

From Leetcoder.

Binary Tree Upside Down

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

For example:
Given a binary tree `{1,2,3,4,5}`,

```
    1
   / \
  2   3
 / \
4   5
```

return the root of the binary tree `[4,5,2,#,#,3,1]`.

```
   4
  / \
 5   2
    / \
   3   1
```

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

OJ's Binary Tree Serialization:
The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

The above binary tree is serialized as `"{1,2,3,#,#,4,#,#,5}"`.

Solution 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

Solution 2

```
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

Solution 3

```
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

From Leetcoder.

Read N Characters Given Read4

The API: `int read4(char *buf)` reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the `read4` API, implement the function `int read(char *buf, int n)` that reads *n* characters from the file.

Note:
The `read` function will only be called once for each test case.

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Read N Characters Given Read4 II - Call multiple times

The API: `int read4(char *buf)` reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the `read4` API, implement the function `int read(char *buf, int n)` that reads *n* characters from the file.

Note:
The `read` function may be called multiple times.

Solution 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

Solution 2

The key is to store memorized variable in the class level and remember offset position and remaining number of elements.

```
/* 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

Solution 3

```
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

From Leetcoder.

Longest Substring with At Most Two Distinct Characters

Given a string, find the length of the longest substring T that contains at most 2 distinct characters.

For example, Given s = `“eceba”`,

T is "ece" which its length is 3.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Intersection of Two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

```
A:          a1 → a2
                   ↘
                     c1 → c2 → c3
                   ↗
B:     b1 → b2 → b3
```

begin to intersect at node c1.

Notes:

- If the two linked lists have no intersection at all, return `null`.
- The linked lists must retain their original structure after the function returns.
- You may assume there are no cycles anywhere in the entire linked structure.
- Your code should preferably run in O(n) time and use only O(1) memory.

Credits:
Special thanks to @stellari for adding this problem and creating all test cases.

Solution 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

Solution 2

I found most solutions here preprocess linkedlists to get the difference in len. Actually we don't care about the "value" of difference, we just want to make sure two pointers reach the intersection node at the same time.

We can use two iterations to do that. In the first iteration, we will reset the pointer of one linkedlist to the head of another linkedlist after it reaches the tail node. In the second iteration, we will move two pointers until they points to the same node. Our operations in first iteration will help us counteract the difference. So if two linkedlist intersects, the meeting point in second iteration must be the intersection point. If the two linked lists have no intersection at all, then the meeting pointer in second iteration must be the tail node of both lists, which is null

Below is my commented Java code:

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    //boundary check
    if(headA == null || headB == null) return null;

    ListNode a = headA;
    ListNode b = headB;

    //if a & b have different len, then we will stop the loop after second iterat
ion
    while( a != b){
        //for the end of first iteration, we just reset the pointer to the head of
another linkedlist
        a = a == null? headB : a.next;
        b = b == null? headA : b.next;
    }

    return a;
}
```

written by hpplayer original link here

Solution 3

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

From Leetcoder.

One Edit Distance

Given two strings S and T, determine if they are both one edit distance apart.

Solution 1

```
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

Solution 2

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

Solution 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

From Leetcoder.

Find Peak Element

A peak element is an element that is greater than its neighbors.

Given an input array where `num[i] ≠ num[i+1]`, find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that `num[-1] = num[n] = -∞`.

For example, in array `[1, 2, 3, 1]`, 3 is a peak element and your function should return the index number 2.

click to show spoilers.

Note:
Your solution should be in logarithmic complexity.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

I find it useful to reason about binary search problems using invariants. While there are many solutions posted here, neither of them provide (in my opinion) a good explanation about why they work. I just spent some time thinking about this and I thought it might be a good idea to share my thoughts.

Assume we initialize left = 0, right = nums.length - 1. The invariant I'm using is the following:

nums[left - 1] < nums[left] && nums[right] > nums[right + 1]

That basically means that in the current interval we're looking, [left, right] the function started increasing to left and will eventually decrease at right. The behavior between [left, right] falls into the following 3 categories:

1) nums[left] > nums[left + 1]. From the invariant, nums[left - 1] < nums[left] => left is a peak

2) The function is increasing from left to right i.e. nums[left] < nums[left + 1] < .. < nums[right - 1] < nums[right]. From the invariant, nums[right] > nums[right + 1] => right is a peak

3) the function increases for a while and then decreases (in which case the point just before it starts decreasing is a peak) e.g. 2 5 6 3 (6 is the point in question)

As shown, if the invariant above holds, there is at least a peak between [left, right]. Now we need to show 2 things:

I) the invariant is initially true. Since left = 0 and right = nums.length - 1 initially and we know that nums[-1] = nums[nums.length] = -oo, this is obviously true

II) At every step of the loop the invariant gets reestablished. If we consider the code in the loop, we have mid = (left + right) / 2 and the following 2 cases:

a) nums[mid] < nums[mid + 1]. It turns out that the interval [mid + 1, right] respects the invariant (nums[mid] < nums[mid + 1] -> part of the cond + nums[right] > nums[right + 1] -> part of the invariant in the previous loop iteration)

b) nums[mid] > nums[mid + 1]. Similarly, [left, mid] respects the invariant (nums[left - 1] < nums[left] -> part of the invariant in the previous loop iteration and nums[mid] > nums[mid + 1] -> part of the cond)

As a result, the invariant gets reestablished and it will also hold when we exit the loop. In that case we have an interval of length 2 i.e. right = left + 1. If nums[left] > nums[right], using the invariant (nums[left - 1] < nums[left]), we get that left is a peak. Otherwise right is the peak (nums[left] < nums[right] and nums[right] < nums[right + 1] from the invariant).

```java
public int findPeakElement(int[] nums) {
    int N = nums.length;
    if (N == 1) {
        return 0;
    }

    int left = 0, right = N - 1;
    while (right - left > 1) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < nums[mid + 1]) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }

    return (left == N - 1 || nums[left] > nums[left + 1]) ? left : right;
}
```

I hope this makes things clear despite the long explanation.

written by cosmin79 original link here

From Leetcoder.

Missing Ranges

Given a sorted integer array where the range of elements are [*lower*, *upper*] inclusive, return its missing ranges.

For example, given `[0, 1, 3, 50, 75]`, *lower* = 0 and *upper* = 99, return `["2", "4->49", "51->74", "76->99"].`

Solution 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

Solution 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

Solution 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

From Leetcoder.

Maximum Gap

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space.

Return 0 if the array contains less than 2 elements.

You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

Credits:
Special thanks to @porker2008 for adding this problem and creating all test cases.

Solution 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

Solution 2

You can look at radix sort visualization here before reading the code: https://www.cs.usfca.edu/~galles/visualization/RadixSort.html

```
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

Solution 3

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

From Leetcoder.

Compare Version Numbers

Compare two version numbers *version1* and *version2*.
If *version1* > *version2* return 1, if *version1* < *version2* return -1, otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the `.` character.
The `.` character does not represent a decimal point and is used to separate number sequences.
For instance, `2.5` is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

Here is an example of version numbers ordering:

```
0.1 < 1.1 < 1.2 < 13.37
```

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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".

```
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

From Leetcoder.

Fraction to Recurring Decimal

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

- Given numerator = 1, denominator = 2, return "0.5".
- Given numerator = 2, denominator = 1, return "2".
- Given numerator = 2, denominator = 3, return "0.(6)".

Credits:
Special thanks to @Shangrila for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Two Sum II - Input array is sorted

Given an array of integers that is already ***sorted in ascending order***, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

```
Input: numbers={2, 7, 11, 15}, target=9
Output: index1=1, index2=2
```

Solution 1

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.

```
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

Solution 2

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

Solution 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

From Leetcoder.

Excel Sheet Column Title

Given a positive integer, return its corresponding column title as appear in an Excel sheet.

For example:

```
1 -> A
2 -> B
3 -> C
...
26 -> Z
27 -> AA
28 -> AB
```

Credits:
Special thanks to @ifanchu for adding this problem and creating all test cases.

Solution 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

Solution 2

```java
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

Solution 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

From Leetcoder.

Majority Element

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

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

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Two Sum III - Data structure design

Design and implement a TwoSum class. It should support the following operations: `add` and `find`.

`add` - Add the number to an internal data structure.
`find` - Find if there exists any pair of numbers which sum is equal to the value.

For example,

```
add(1); add(3); add(5);
find(4) -> true
find(7) -> false
```

Solution 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:

```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

Solution 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

Solution 3

```
public class TwoSum {
List<Integer> list = new ArrayList<Integer>();
Map<Integer, Integer> map = new HashMap<Integer, Integer>();

// Add the number to an internal data structure.
public void add(int number) {
    list.add(number);
    if (map.containsKey(number))
        map.put(number, map.get(number)+1);
    else
        map.put(number, 1);
}

// Find if there exists any pair of numbers which sum is equal to the value.
public boolean find(int value) {
    for (int i = 0; i < list.size(); i++) {
        int num1 = list.get(i);
        int num2 = value - num1;
        if ((num1 == num2 && map.get(num1) > 1) || (num1 != num2 && map.containsK
ey(num2)))
            return true;
    }
    return false;
}
```

}

written by whdawn original link here

From Leetcoder.

Excel Sheet Column Number

Related to question Excel Sheet Column Title

Given a column title as appear in an Excel sheet, return its corresponding column number.

For example:

```
A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
```

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Factorial Trailing Zeroes

Given an integer n, return the number of trailing zeroes in $n!$.

Note: Your solution should be in logarithmic time complexity.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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

Solution 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 4×5 = 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 ÷ 5 = 20, so, there are twenty multiples of 5 between 1 and 100.

but wait, actually 25 is 5×5, so each multiple of 25 has an extra factor of 5, e.g. 25 × 4 = 100， which introduces extra of zero.

So, we need know how many multiples of 25 are between 1 and 100? Since 100 ÷ 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, 5×5, 5×5×5, 5×5×5×5 ....

Example Three

By given number 4617.

5^1 : 4617 ÷ 5 = 923.4, so we get 923 factors of 5

5^2 : 4617 ÷ 25 = 184.68, so we get 184 additional factors of 5

5^3 : 4617 ÷ 125 = 36.936, so we get 36 additional factors of 5

5^4 : 4617 ÷ 625 = 7.3872, so we get 7 additional factors of 5

5^5 : 4617 ÷ 3125 = 1.47744, so we get 1 more factor of 5

5^6 : 4617 ÷ 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

Solution 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

From Leetcoder.

Binary Search Tree Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling `next()` will return the next smallest number in the BST.

Note: `next()` and `hasNext()` should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Dungeon Game

The demons had captured the princess (**P**) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (**K**) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (*negative* integers) upon entering these rooms; other rooms are either empty (*0's*) or contain magic orbs that increase the knight's health (*positive* integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.

For example, given the dungeon below, the initial health of the knight must be at least 7 if he follows the optimal path `RIGHT-> RIGHT -> DOWN -> DOWN`.

-2 (K)	-3	3
-5	-10	1
10	30	-5 (P)

Notes:

- The knight's health has no upper bound.
- Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

Credits:
Special thanks to @stellari for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Largest Number

Given a list of non negative integers, arrange them such that they form the largest number.

For example, given `[3, 30, 34, 5, 9]`, the largest formed number is `9534330`.

Note: The result may be very large, so you need to return a string instead of an integer.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Reverse Words in a String II

Given an input string, reverse the string word by word. A word is defined as a sequence of non-space characters.

The input string does not contain leading or trailing spaces and the words are always separated by a single space.

For example,
Given s = "`the sky is blue`",
return "`blue is sky the`".

Could you do it *in-place* without allocating extra space?

Related problem: Rotate Array

Solution 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

Solution 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

Solution 3

```
class Solution {
public:
    void reverseWords(string &s) {
        vector<int> pos;
        pos.push_back(-1);
        for (int i = 0; i < s.size(); i++) if (s[i] == ' ') pos.push_back(i);
        pos.push_back(s.size());
        for (int i = pos.size() - 2; i >= 0; i--) reverse(s.begin() + pos[i] + 1,
s.begin() + pos[i + 1]);
        reverse(s.begin(), s.end());
    }
};
```

written by orzzzl original link here

From Leetcoder.

Repeated DNA Sequences

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

For example,

```
Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",

Return:
["AAAAACCCCC", "CCCCCAAAAA"].
```

Solution 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

Solution 2

```
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

Solution 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

From Leetcoder.

Best Time to Buy and Sell Stock IV

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most **k** transactions.

Note:
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

```
/**
 * dp[i, j] represents the max profit up until prices[j] using at most i transacti
ons.
 * dp[i, j] = max(dp[i, j-1], prices[j] - prices[jj] + dp[i-1, jj]) { jj in range
of [0, j-1] }
 *          = max(dp[i, j-1], prices[j] + max(dp[i-1, jj] - prices[jj]))
 * dp[0, j] = 0; 0 transactions makes 0 profit
 * dp[i, 0] = 0; if there is only one price data point you can't make any transact
ion.
 */

public int maxProfit(int k, int[] prices) {
    int n = prices.length;
    if (n <= 1)
        return 0;

    //if k >= n/2, then you can make maximum number of transactions.
    if (k >=  n/2) {
        int maxPro = 0;
        for (int i = 1; i < n; i++) {
            if (prices[i] > prices[i-1])
                maxPro += prices[i] - prices[i-1];
        }
        return maxPro;
    }

    int[][] dp = new int[k+1][n];
    for (int i = 1; i <= k; i++) {
        int localMax = dp[i-1][0] - prices[0];
        for (int j = 1; j < n; j++) {
            dp[i][j] = Math.max(dp[i][j-1],  prices[j] + localMax);
            localMax = Math.max(localMax, dp[i-1][j] - prices[j]);
        }
    }
    return dp[k][n-1];
}
```

written by jinwu original link here

From Leetcoder.

Rotate Array

Rotate an array of n elements to the right by k steps.

For example, with $n = 7$ and $k = 3$, the array `[1,2,3,4,5,6,7]` is rotated to `[5,6,7,1,2,3,4]`.

Note:
Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

[show hint]

Hint:
Could you do it in-place with O(1) extra space?
Related problem: Reverse Words in a String II

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Reverse Bits

Reverse bits of a given 32 bits unsigned integer.

For example, given input 43261596 (represented in binary as **00000010100101000001111010011100**), return 964176192 (represented in binary as **00111001011110000010100101000000**).

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

Related problem: Reverse Integer

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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

Solution 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.

```
// 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

Solution 3

```
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

From Leetcoder.

Number of 1 Bits

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer '11' has binary representation `00000000000000000000000000001011`, so the function should return 3.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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

Solution 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)
}
```

——From Wikipedia.

written by makuiyu original link here

Solution 3

```
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

From Leetcoder.

House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

Credits:
Special thanks to @ifanchu for adding this problem and creating all test cases. Also thanks to @ts for adding additional test cases.

Solution 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

Solution 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

Solution 3

```
public int rob(int[] num) {
    int rob = 0; //max monney can get if rob current house
    int notrob = 0; //max money can get if not rob current house
    for(int i=0; i<num.length; i++) {
        int currob = notrob + num[i]; //if rob current value, previous house must
not be robbed
        notrob = Math.max(notrob, rob); //if not rob ith house, take the max valu
e of robbed (i-1)th house and not rob (i-1)th house
        rob = currob;
    }
    return Math.max(rob, notrob);
}
```

written by wenwen original link here

From Leetcoder.

Binary Tree Right Side View

Given a binary tree, imagine yourself standing on the *right* side of it, return the values of the nodes you can see ordered from top to bottom.

For example:
Given the following binary tree,

```
   1
```

```
You should return
[1, 3, 4]
.
```

Credits:
Special thanks to @amrsaqr for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

```
class Solution {
public:
    void dfs(TreeNode* root, int lv, vector<int> &res){
        if(!root)   return;
        if(lv>=res.size()) res.push_back(root->val);
        dfs(root->right,lv+1,res);
        dfs(root->left,lv+1,res);
    }

    vector<int> rightSideView(TreeNode* root) {
        vector<int> res;
        dfs(root, 0, res);
        return res;
    }
};
```

written by sjtuldk original link here

From Leetcoder.

Number of Islands

Given a 2d grid map of `'1'`s (land) and `'0'`s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
11110
11010
11000
00000
```

Answer: 1

Example 2:

```
11000
11000
00100
00011
```

Answer: 3

Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.

Solution 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

Solution 2

```
public class Solution {

private int n;
private int m;

public int numIslands(char[][] grid) {
    int count = 0;
    m = grid.length;
    if (m == 0) return 0;
    n = 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

Solution 3

Sink and count the islands.

Python Solution

```python
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

From Leetcoder.

Bitwise AND of Numbers Range

Given a range [m, n] where 0

For example, given the range [5, 7], you should return 4.

Credits:
Special thanks to @amrsaqr for adding this problem and creating all test cases.

Solution 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

Solution 2

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

Solution 3

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

From Leetcoder.

Happy Number

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

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

Example: 19 is a happy number

$1^2 + 9^2 = 82$
$8^2 + 2^2 = 68$
$6^2 + 8^2 = 100$
$1^2 + 0^2 + 0^2 = 1$

Credits:
Special thanks to @mithmatt and @ts for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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 m010 original link here

From Leetcoder.

Remove Linked List Elements

Remove all elements from a linked list of integers that have value ***val***.

Example
Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, ***val*** = 6
Return: 1 --> 2 --> 3 --> 4 --> 5

Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.

Solution 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

Solution 2

```java
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

Solution 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

From Leetcoder.

Count Primes

Description:

Count the number of prime numbers less than a non-negative number, $\boldsymbol{n}$.

Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.

1. Let's start with a *isPrime* function. To determine if a number is prime, we need to check if it is not divisible by any number less than n. The runtime complexity of *isPrime* function would be $O(n)$ and hence counting the total prime numbers up to n would be $O(n^2)$. Could we do better?

2. As we know the number must not be divisible by any number $> n / 2$, we can immediately cut the total iterations half by dividing only up to $n / 2$. Could we still do better?

3. Let's write down all of 12's factors:

   ```
   2 × 6 = 12
   3 × 4 = 12
   4 × 3 = 12
   6 × 2 = 12
   ```

   As you can see, calculations of 4×3 and 6×2 are not necessary. Therefore, we only need to consider factors up to $\sqrt{n}$ because, if n is divisible by some number p, then $n = p \times q$ and since $p \leq q$, we could derive that $p \leq \sqrt{n}$.

   Our total runtime has now improved to $O(n^{1.5})$, which is slightly better. Is there a faster approach?

   ```
   public int countPrimes(int n) {
      int count = 0;
      for (int i = 1; i
   ```

4. The Sieve of Eratosthenes is one of the most efficient ways to find all prime numbers up to n. But don't let that name scare you, I promise that the concept is surprisingly simple.

Sieve of Eratosthenes: algorithm steps for primes below 121. " Sieve of Eratosthenes Animation" by SKopp is licensed under CC BY 2.0.

We start off with a table of n numbers. Let's look at the first number, 2. We know all multiples of 2 must not be primes, so we mark them off as non-primes. Then we look at the next number, 3. Similarly, all multiples of 3 such as $3 \times 2 = 6$, $3 \times 3 = 9$, ... must not be primes, so we mark them off as well. Now we look at the next number, 4, which was already marked off. What does this tell you? Should you mark off all multiples of 4 as well?

5. 4 is not a prime because it is divisible by 2, which means all multiples of 4 must also be divisible by 2 and were already marked off. So we can skip 4 immediately and go to the next number, 5. Now, all multiples of 5 such as $5 \times 2 = 10$, $5 \times 3 = 15$, $5 \times 4 = 20$, $5 \times 5 = 25$, ... can be marked off. There is a slight optimization here, we do not need to start from $5 \times 2 = 10$. Where should we start marking off?

6. In fact, we can mark off multiples of 5 starting at $5 \times 5 = 25$, because $5 \times 2 = 10$ was already marked off by multiple of 2, similarly $5 \times 3 = 15$ was already marked off by multiple of 3. Therefore, if the current number is p, we can always mark off multiples of p starting at p^2, then in increments of p: $p^2 + p$, $p^2 + 2p$, ... Now what should be the terminating loop condition?

7. It is easy to say that the terminating loop condition is p n, which is certainly correct but not efficient. Do you still remember *Hint #3*?

8. Yes, the terminating loop condition can be p n, as all non-primes $\geq \sqrt{n}$ must have already been marked off. When the loop terminates, all the numbers in the table that are non-marked are prime.

The Sieve of Eratosthenes uses an extra $O(n)$ memory and its runtime

complexity is O(n log log n). For the more mathematically inclined readers, you can read more about its algorithm complexity on Wikipedia.

```
public int countPrimes(int n) {
   boolean[] isPrime = new boolean[n];
   for (int i = 2; i
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Isomorphic Strings

Given two strings ***s*** and ***t***, determine if they are isomorphic.

Two strings are isomorphic if the characters in***s*** can be replaced to get ***t***.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

For example,
Given `"egg"`, `"add"`, return true.

Given `"foo"`, `"bar"`, return false.

Given `"paper"`, `"title"`, return true.

Note:
You may assume both ***s*** and ***t*** have the same length.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Reverse Linked List

Reverse a singly linked list.

click to show more hints.

Hint:
A linked list can be reversed either iteratively or recursively. Could you implement both?

Solution 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

Solution 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

Solution 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

From Leetcoder.

Course Schedule

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

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

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

For example:

```
2, [[1,0]]
```

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

```
2, [[1,0],[0,1]]
```

There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

Note:
The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.

click to show more hints.

Hints:

1. This problem is equivalent to finding if a cycle exists in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

Solution 1

```java
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

Solution 2

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

Solution 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

From Leetcoder.

Implement Trie (Prefix Tree)

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

Note:
You may assume that all inputs are consist of lowercase letters `a-z`.

Solution 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

Solution 2

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

Solution 3

```
    /**
     ** 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;
                } 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

From Leetcoder.

Minimum Size Subarray Sum

Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a subarray of which the sum ≥ **s**. If there isn't one, return 0 instead.

For example, given the array `[2,3,1,2,4,3]` and `s = 7`,
the subarray `[4,3]` has the minimal length under the problem constraint.

click to show more practice.

More practice:
If you have figured out the $O(n)$ solution, try coding another solution of which the time complexity is $O(n \log n)$.

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

Solution 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

Solution 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.

```
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

Solution 3

```
public int minSubArrayLen(int s, int[] a) {
  if (a == null || a.length == 0)
    return 0;

  int i = 0, j = 0, sum = 0, min = Integer.MAX_VALUE;

  while (j < a.length) {
    sum += a[j++];

    while (sum >= s) {
      min = Math.min(min, j - i);
      sum -= a[i++];
    }
  }

  return min == Integer.MAX_VALUE ? 0 : min;
}
```

written by jeantimex original link here

From Leetcoder.

Course Schedule II

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

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

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

For example:

```
2, [[1,0]]
```

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is `[0,1]`

```
4, [[1,0],[2,0],[3,1],[3,2]]
```

There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is `[0,1,2,3]`. Another correct ordering is `[0,2,1,3]`.

Note:
The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.

click to show more hints.

Hints:

1. This problem is equivalent to finding the topological order in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

Solution 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.

```java
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.

```java
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

Solution 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

```
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

Solution 3

```
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

Design a data structure that supports the following two operations:

```
void addWord(word)
bool search(word)
```

search(word) can search a literal word or a regular expression string containing only letters `a-z` or `.`. A `.` means it can represent any one letter.

For example:

```
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
```

Note:
You may assume that all words are consist of lowercase letters `a-z`.

click to show hint.

You should be familiar with how a Trie works. If not, please work on this problem: Implement Trie (Prefix Tree) first.

Solution 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;
                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

Solution 2

```java
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

Solution 3

Using backtrack to check each character of word to search.

```
public class WordDictionary {
    public class TrieNode {
        public TrieNode[] children = new TrieNode[26];
        public String item = "";
    }

    private TrieNode root = new TrieNode();

    public void addWord(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            if (node.children[c - 'a'] == null) {
                node.children[c - 'a'] = new TrieNode();
            }
            node = node.children[c - 'a'];
        }
        node.item = word;
    }

    public boolean search(String word) {
        return match(word.toCharArray(), 0, root);
    }

    private boolean match(char[] chs, int k, TrieNode node) {
        if (k == chs.length) return !node.item.equals("");
        if (chs[k] != '.') {
            return node.children[chs[k] - 'a'] != null && match(chs, k + 1, node.
children[chs[k] - 'a']);
        } else {
            for (int i = 0; i < node.children.length; i++) {
                if (node.children[i] != null) {
                    if (match(chs, k + 1, node.children[i])) {
                        return true;
                    }
                }
            }
        }
        return false;
    }
}
```

written by Lnic original link here

From Leetcoder.

Word Search II

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

For example,
Given **words** = `["oath","pea","eat","rain"]` and **board** =

```
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]
```

Return `["eat","oath"]`.
Note:
You may assume that all inputs are consist of lowercase letters `a-z`.

click to show hint.

You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?

If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.

Solution 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

Solution 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;
```

```
            char temp;
            if(board[i][j]=='X') return; // visited before;
            if(nullptr == root->children[board[i][j]-'a']) return ; // no string
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
```

```
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] = '*';
        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

Solution 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

From Leetcoder.

House Robber II

Note: This is an extension of House Robber.

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Shortest Palindrome

Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.

For example:

Given `"aacecaaa"`, return `"aaacecaaa"`.

Given `"abcd"`, return `"dcbabcd"`.

Credits:
Special thanks to @ifanchu for adding this problem and creating all test cases. Thanks to @Freezen for additional test cases.

Solution 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

Solution 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

Solution 3

Firstly, let me share my understanding of KMP algorithm. The key of KMP is to build a look up table that records the match result of prefix and postfix. Value in the table means the max len of matching substring that exists in both prefix and postfix. In the prefix this substring should starts from 0, while in the postfix this substring should ends at current index.

For example, now we have a string "ababc" The KMP table will look like this:

> a b a b c
>
> 0 0 1 2 0

(Note: we will not match substring with itself, so we will skip index 0)

So how does this table help us search string match faster?

Well, the answer is if we are trying to match a char after postfix with target string and failed, then we can smartly shift the string, so that the matching string in prefix will replace postfix and now we can try to match the char after prefix with this char in target.

Take above string as an example.

Now we try to match string "ababc" with "abababc".

We will initially have match as below

> a b a b a b c (string x)
>
> a b a b c (string y)
>
> 0 1 2 3 4 5 6

We found char at index 4 does not match, then we can use lookup table and shift the string y wisely. We found table[3] = 2, which means we can shift the string y rightward by 2, and still have same but shorter prefix before index 4, like this:

> a b a b a b c (string x)
>
> ___a b a b c (string y)
>
> 0 1 2 3 4 5 6

If there is a long gap between prefix and postfix, this shift can help us save a lot of time. In the brute force way, we cannot do that because we have no information of the string. We have to compare each possible pair of chars. While in kmp, we know the information of string y so we can move smartly. We can directly jump to the next possible matching pair while discard useless pair of chars.

We are almost done with KMP, but we still have one special case that needs to be

taken care of.

Say now we have a input like this:

> a a b a a a (input String)
>
> 0 1 2 3 4 5 (index)
>
> 0 1 0 1 2 ? (KMP table)

How should we build the KMP table for this string?

Say the pointer in prefix is "x", which is at index 2 now and the pointer in postfix is "y" which is at index 5 now. we need to match "b" pointed by x with "a" pointed by y. It is an unmatched pair, how should we update the cell?

Well, we really don't need to reset it to 0, that will make us skip a valid shorter matching substring "aa". What we do now is just to shorten the length of substring by 1 unit and try to match a shorter substring "aa". This can be done by moving pointer x to the index recorded in [indexOf(x)-1] while keep pointer y stay still. This is because by following the value in KMP table we can always make sure previous part of prefix and postfix is matched even we have shorten their length, so we only need to care about the char after matched part in prefix and postfix.

Use above example:

Firstly we try to compare prefix "aab" with postfix "aaa", pointer in prefix now points to "b" while pointer in postfix now points to "a". So this means current len of postfix/prefix will not give a match, we need to shorten it.

So in the second step, we will fix pointer in postfix, and move pointer in prefix so that we can compare shorter prefix and postfix. The movement of pointer in prefix (say at index x) is done by using KMP table. We will set pointer in prefix to be table [indexOf(x)-1]. In this case, we will move prefix pointer to index 1. So now we try to compare prefix "aa" with postfix "aa".

Finally, we found the matching prefix and postfix, we just update the cell accordingly.

Above is my understanding of KMP algorithm, so how could we apply KMP to this problem

========== I am just a splitter ==========

This problem asks us to add string before the input so the result string will be a palindrome. We can convert it to an alternative problem"find the longest palindrome substring starts from index 0". If we can get the length of such substring, then we can easily build a palindrome string by inserting the reverse part of substring after such substring before the original string.

Example:

input string:

> abacd

longest palindrome substring starts from 0:

> aba

Insert the reverse part of substring after palindrome substring before the head:

> dcabacd

Now the problem becomes how to find the longest palindrome substring starts from 0. We can solve it by using a trick + KMP.

The trick is to build a temp string like this:

> s + "#" + reverse(s)

Then we run KMP on it, the value in last cell will be our solution. In this problem, we don't need to use KMP to match strings but instead we use the lookup table in KMP to find the palindrome.

We add "#" here to force the match in reverse(s) starts from its first index What we do in KMP here is trying to find a match between prefix in s and a postfix in reverse(s). The match part will be palindrome substring.

Example: input:

> catacb

Temp String:

> catacb # bcatac

KMP table:

> c a t a c b # b c a t a c
>
> 0 0 0 0 1 0 0 0 1 2 3 4 5

In the last cell, we got a value 5. It means in s we have a substring of length 5 that is palindrome.

So, above is my understanding of KMP any solution towards this problem. Below is my code

```java
public String shortestPalindrome(String s) {
    String temp = s + "#" + new StringBuilder(s).reverse().toString();
    int[] table = getTable(temp);

    //get the maximum palin part in s starts from 0
    return new StringBuilder(s.substring(table[table.length - 1])).reverse().toSt
ring() + s;
}

public int[] getTable(String s){
    //get lookup table
    int[] table = new int[s.length()];

    //pointer that points to matched char in prefix part

    int index = 0;
    //skip index 0, we will not match a string with itself
    for(int i = 1; i < s.length(); i++){
        if(s.charAt(index) == s.charAt(i)){
            //we can extend match in prefix and postfix
            table[i] = table[i-1] + 1;
            index ++;
        }else{
            //match failed, we try to match a shorter substring

            //by assigning index to table[i-1], we will shorten the match string
length, and jump to the
            //prefix part that we used to match postfix ended at i - 1
            index = table[i-1];

            while(index > 0 && s.charAt(index) != s.charAt(i)){
                //we will try to shorten the match string length until we revert
to the beginning of match (index 1)
                index = table[index-1];
            }

            //when we are here may either found a match char or we reach the boun
dary and still no luck
            //so we need check char match
            if(s.charAt(index) == s.charAt(i)){
                //if match, then extend one char
                index ++ ;
            }

            table[i] = index;
        }

    }

    return table;
}
```

If I messed up or misunderstood something, please leave comment below. Thanks ~ written by hpplayer original link here

From Leetcoder.

Kth Largest Element in an Array

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

For example,
Given `[3,2,1,5,6,4]` and k = 2, return 5.

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

Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.

Solution 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).

```java
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.

```java
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

Solution 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

Solution 3

https://en.wikipedia.org/wiki/Quickselect

```
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

From Leetcoder.

Combination Sum III

Find all possible combinations of ***k*** numbers that add up to a number ***n***, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Ensure that numbers within the set are sorted in ascending order.

Example 1:

Input: ***k*** = 3, ***n*** = 7

Output:

```
[[1,2,4]]
```

Example 2:

Input: ***k*** = 3, ***n*** = 9

Output:

```
[[1,2,6], [1,3,5], [2,3,4]]
```

Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

```
 public List<List<Integer>> combinationSum3(int k, int n) {
    List<List<Integer>> ans = new ArrayList<>();
    combination(ans, new ArrayList<Integer>(), k, 1, n);
    return ans;
}

private void combination(List<List<Integer>> ans, List<Integer> comb, int k,  int
start, int n) {
    if (comb.size() == k && n == 0) {
        List<Integer> li = new ArrayList<Integer>(comb);
        ans.add(li);
        return;
    }
    for (int i = start; i <= 9; i++) {
        comb.add(i);
        combination(ans, comb, k, i+1, n-i);
        comb.remove(comb.size() - 1);
    }
}
```

written by jinwu original link here

From Leetcoder.

Contains Duplicate

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

Solution 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.

```java
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

Solution 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

Solution 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

From Leetcoder.

The Skyline Problem

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are **given the locations and height of all the buildings** as shown on a cityscape photo (Figure A), write a program to **output the skyline** formed by these buildings collectively (Figure B).

The geometric information of each building is represented by a triplet of integers `[Li, Ri, Hi]`, where `Li` and `Ri` are the x coordinates of the left and right edge of the ith building, respectively, and `Hi` is its height. It is guaranteed that `0 ≤ Li, Ri ≤ INT_MAX`, `0 , and  Ri - Li > 0`. `You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.`

For instance, the dimensions of all buildings in Figure A are recorded as: `[ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ]` .

The output is a list of "**key points**" (red dots in Figure B) in the format of `[ [x1,y1], [x2, y2], [x3, y3], ... ]` that uniquely defines a skyline. **A key point is the left endpoint of a horizontal line segment**. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the skyline in Figure B should be represented as: `[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]`.

Notes:

- The number of buildings in any input list is guaranteed to be in the range `[0, 10000]`.
- The input list is already sorted in ascending order by the left x position `Li`.
- The output list must be sorted by the x position.
- There must be no consecutive horizontal lines of equal height in the output skyline. For instance, `[...[2 3], [4 5], [7 5], [11 5], [12 7]...]` is not acceptable; the three lines of height 5 should be merged into one in the final output as such: `[...[2 3], [4 5], [12 7], ...]`

Credits:
Special thanks to @stellari for adding this problem, creating these two awesome images and all test cases.

Solution 1

```
    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

Solution 2

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

Solution 3

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

Contains Duplicate II

Given an array of integers and an integer *k*, find out whether there are two distinct indices *i* and *j* in the array such that **nums[i] = nums[j]** and the difference between *i* and *j* is at most *k*.

Solution 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

Solution 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

Solution 3

```
public boolean containsNearbyDuplicate(int[] nums, int k) {
    Map<Integer, Integer> map = new HashMap<Integer, Integer>();
    for (int i = 0; i < nums.length; i++) {
        if (map.containsKey(nums[i])) {
            if (i - map.get(nums[i]) <= k) return true;
        }
        map.put(nums[i], i);
    }
    return false;
}
```

written by wz366 original link here

From Leetcoder.

Contains Duplicate III

Given an array of integers, find out whether there are two distinct indices *i* and *j* in the array such that the difference between **nums[i]** and **nums[j]** is at most *t* and the difference between *i* and *j* is at most *k*.

Solution 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

Solution 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

Solution 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
        // |x - nums[i]| <= t  ==> -t <= x - nums[i] <= t;
        auto pos = window.lower_bound(nums[i] - t); // x-nums[i] >= -t ==> x >= n
ums[i]-t
        // x - nums[i] <= t ==> |x - nums[i]| <= t
        if (pos != window.end() && *pos - nums[i] <= t) return true;
        window.insert(nums[i]);
    }
    return false;
}
```

written by lchen77 original link here

From Leetcoder.

Maximal Square

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area.

For example, given the following matrix:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

Return 4.

Credits:

Special thanks to @Freezen for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Count Complete Tree Nodes

Given a **complete** binary tree, count the number of nodes.

Definition of a complete binary tree from Wikipedia:
In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h.

Solution 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

Solution 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

Solution 3

public class Solution {

```
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

From Leetcoder.

Rectangle Area

Find the total area covered by two **rectilinear** rectangles in a **2D** plane.

Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.

Assume that the total area is never beyond the maximum possible value of **int**.

Credits:
Special thanks to @mithmatt for adding this problem, creating the above image and all test cases.

Solution 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

Solution 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

Solution 3

```
This is utterly ridiculous. As I was writing this I knew I was going about it wro
ng but I wanted to finish it before I thought of a different method or looked at
any other solutions.

  int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {
    int area1 = (D - B)*(C - A);
    int area2 = (H - F)*(G - E);
    int area3;
    if (area1 == 0) {
        return area2;
    }
    if (area2 == 0) {
        return area1;
    }
    if ((A == D) && (B == F) && (C == G) && (D == H)) {
        return area1;
    }
    if ((E >= C) | (G <= A) | (H <= B) | (D <= F)) {    //not overlapping
        return (area1 + area2);
    }
    if (((G - E) <= (C - A)) && ((H - F) <= (D - B)) && (E >= A) && (F >= B) && (G
<= C) && (D >= H)) {                        //rect2 is inside rect1
        return area1;
    }
    if (((C - A) <= (G - E)) && ((D - B) <= (H - F)) && (E <= A) && (B >= F) && (G
>= C) && (H >= D)) {                        //rect1 is inside rect2
        return area2;
    }
    if ((F >= B) && (E >= A) && (G >= C) && (H >= D)) {                       //ov
erlapping upper right corner
        area3 = (C - E)*(D - F);
    }
    else if ((F >= B) && (E <= A) && (G <= C) && (H >= D)) {
//overlapping upper left corner
        area3 = (G - A)*(D - F);
    }
    else if ((F <= B) && (E <= A) && (G <= C) && (H <= D)) {
//overlapping bottom left corner
        area3 = (G - A)*(H - B);
    }
    else if ((F <= B) && (E >= A) && (G >= C) && (H <= D)) {
//overlapping bottom right corner
        area3 = (H - B)*(C - E);
    }
    else if (((C - A) <= (G - E)) && (H <= D) && (G >= C) && (E <= A) && (F <= B))
{               //overlapping bottom side
        area3 = (C - A)*(H - B);
    }
    else if (((C - A) <= (G - E)) && (H >= D) && (G >= C) && (E <= A) && (F >= B))
{               //overlapping top side
        area3 = (C - A)*(D - F);
    }
    else if (((D - B) <= (H - F)) && (E <= A) && (F <= B) && (H >= D) && (G <= C))
{               //overlapping left side
```

```
        area3 = (G - A)*(D - B);
    }
    else if (((D - B) <= (H - F)) && (E >= A) && (F <= B) && (H >= D) && (G >= C))
{               //overlapping right side
        area3 = (C - E)*(D - B);
    }
    else if (((C - A) >= (G - E)) && (E >= A) && (F >= B) && (C >= G) && (D <= H))
{      //overlapping part of top side
        area3 = (G - E)*(D - F);
    }
    else if (((C - A) >= (G - E)) && (A <= E) && (B >= F) && (G <= C) && (D >= H))
{       //overlapping part of bottom side
        area3 = (G - E)*(H - B);
    }
    else if (((D - B) >= (H - F)) && (E <= A) && (F >= B) && (G <= C) && (H <= D))
{      //overlapping part of left side
        area3 = (G - A)*(H - F);
    }
    else if (((D - B) >= (H - F)) && (E >= A) && (F >= B) && (G >= C) && (H <= D))
{       //overlapping part of right side
        area3 = (C - E)*(H - F);
    }
    else if (((G - E) <= (C - A)) && (E >= A) && (F <= B) && (G <= C) && (H >= D))
{     //overlapping top and bottom
        area3 = (G - E)*(D - B);
    }
    else if (((H - F) <= (D - B)) && (E <= A) && (F >= B) && (C <= G) && (D >= H))
{     //overlapping left and right
        area3 = (C - A)*(H - F);
    }

    return (area1 + area2 - area3);
}
```

written by bigDeeOT original link here

From Leetcoder.

Basic Calculator

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open `(` and closing parentheses `)`, the plus `+` or minus sign `-`, **non-negative** integers and empty spaces .

You may assume that the given expression is always valid.

Some examples:

```
"1 + 1" = 2
" 2-1 + 2 " = 3
"(1+(4+5+2)-3)+(6+8)" = 23
```

Note: Do not use the `eval` built-in library function.

Solution 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

Solution 2

```
class Solution {
public:
    int calculate(string s) {
        stack <int> nums, ops;
        int num = 0;
        int rst = 0;
        int sign = 1;
        for (char c : s) {
            if (isdigit(c)) {
                num = num * 10 + c - '0';
            }
            else {
                rst += sign * num;
                num = 0;
                if (c == '+') sign = 1;
                if (c == '-') sign = -1;
                if (c == '(') {
                    nums.push(rst);
                    ops.push(sign);
                    rst = 0;
                    sign = 1;
                }
                if (c == ')' && ops.size()) {
                    rst = ops.top() * rst + nums.top();
                    ops.pop(); nums.pop();
                }
            }
        }
        rst += sign * num;
        return rst;
    }
};
```

written by d40a original link here

Solution 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

From Leetcoder.

Implement Stack using Queues

Implement the following operations of a stack using queues.

- push(x) -- Push element x onto stack.
- pop() -- Removes the element on top of the stack.
- top() -- Get the top element.
- empty() -- Return whether the stack is empty.

Notes:

- You must use *only* standard operations of a queue -- which means only `push to back`, `peek/pop from front`, `size`, and `is empty` operations are valid.
- Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue.
- You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

Update (2015-06-11):
The class name of the **Java** function had been updated to **MyStack** instead of Stack.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and all test cases.

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Invert Binary Tree

Invert a binary tree.

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

to

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

Trivia:
This problem was inspired by this original tweet by Max Howell:

> Google: 90% of our engineers use the software you wrote (Homebrew), but you can't invert a binary tree on a whiteboard so fuck off.

Solution 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

Solution 2

Recursive

```
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

Solution 3

```
def invertTree(self, root):
    if root:
        root.left, root.right = self.invertTree(root.right), self.invertTree(root
.left)
        return root
```

Maybe make it four lines for better readability:

```
def invertTree(self, root):
    if root:
        invert = self.invertTree
        root.left, root.right = invert(root.right), invert(root.left)
        return root
```

And an iterative version using my own stack:

```
def invertTree(self, root):
    stack = [root]
    while stack:
        node = stack.pop()
        if node:
            node.left, node.right = node.right, node.left
            stack += node.left, node.right
    return root
```

written by StefanPochmann original link here

From Leetcoder.

Basic Calculator II

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only**non-negative** integers, `+`, `-`, `*`, `/` operators and empty spaces . The integer division should truncate toward zero.

You may assume that the given expression is always valid.

Some examples:

```
"3+2*2" = 7
" 3/2 " = 1
" 3+5 / 2 " = 5
```

Note: Do not use the `eval` built-in library function.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

Given a sorted integer array without duplicates, return the summary of its ranges.

For example, given `[0,1,2,4,5,7]`, return `["0->2","4->5","7"].`

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 2

```
   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

Solution 3

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

From Leetcoder.

Majority Element II

Given an integer array of size n, find all elements that appear more than `⌊ n/3 ⌋` times. The algorithm should run in linear time and in O(1) space.

1. How many majority elements could it possibly have?
2. Do you have a better hint? Suggest it!

Solution 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

Solution 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

Solution 3

Boyer-Moore Majority Vote algorithm generalization to elements appear more than floor(n/k) times

```
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

From Leetcoder.

Kth Smallest Element in a BST

Given a binary search tree, write a function `kthSmallest` to find the **k**th smallest element in it.

Note:
You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

Follow up:
What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

1. Try to utilize the property of a BST.
2. What if you could modify the BST node's structure?
3. The optimal runtime complexity is O(height of BST).

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 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:

```java
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

Solution 2

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.

```
   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

Solution 3

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

From Leetcoder.

Power of Two

Given an integer, write a function to determine if it is a power of two.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Implement Queue using Stacks

Implement the following operations of a queue using stacks.

- push(x) -- Push element x to the back of queue.
- pop() -- Removes the element from in front of queue.
- peek() -- Get the front element.
- empty() -- Return whether the queue is empty.

Notes:

- You must use *only* standard operations of a stack -- which means only `push to top`, `peek/pop from top`, `size`, and `is empty` operations are valid.
- Depending on your language, stack may not be supported natively. You may simulate a stack by using a list or deque (double-ended queue), as long as you use only standard operations of a stack.
- You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue).

Solution 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

```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++

```cpp
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

Solution 2

```
class MyQueue {
Stack<Integer> queue = new Stack<Integer>();
// Push element x to the back of queue.
public void push(int x) {
    Stack<Integer> temp = new Stack<Integer>();
    while(!queue.empty()){
        temp.push(queue.pop());
    }
    queue.push(x);
    while(!temp.empty()){
        queue.push(temp.pop());
    }
}

// Removes the element from in front of queue.
public void pop() {
    queue.pop();
}

// Get the front element.
public int peek() {
    return queue.peek();
}

// Return whether the queue is empty.
public boolean empty() {
    return queue.empty();
}
```

}

written by yangneu2015 original link here

Solution 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

From Leetcoder.

Number of Digit One

Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n.

For example:
Given n = 13,
Return 6, because digit 1 occurred in the following numbers: 1, 10, 11, 12, 13.

1. Beware of overflow.

Solution 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

Solution 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

Solution 3

For every digit in n (Suppose n = 240315, the digits are 2, 4, 0, 3, 1, 5)， I 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

Given a singly linked list, determine if it is a palindrome.

Follow up:
Could you do it in O(n) time and O(1) space?

Solution 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

Solution 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

Solution 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

From Leetcoder.

Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”

```
       _______6______
      /              \
   ___2__          ___8__
  /      \        /      \
  0      _4       7       9
        /  \
        3   5
```

For example, the lowest common ancestor (LCA) of nodes `2` and `8` is `6`. Another example is LCA of nodes `2` and `4` is `2`, since a node can be a descendant of itself according to the LCA definition.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”

```
       _______3______
      /              \
   ___5__          ___1__
  /      \        /      \
  6      _2       0       8
        /  \
        7   4
```

For example, the lowest common ancestor (LCA) of nodes `5` and `1` is `3`. Another example is LCA of nodes `5` and `4` is `5`, since a node can be a descendant of itself according to the LCA definition.

Solution 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

```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

Solution 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

Solution 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

From Leetcoder.

Delete Node in a Linked List

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Supposed the linked list is `1 -> 2 -> 3 -> 4` and you are given the third node with value `3`, the linked list should become `1 -> 2 -> 4` after calling your function.

Solution 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

Solution 2

This question is stupid and should be deleted intermediately.

written by smfwuxiao original link here

Solution 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

From Leetcoder.

Product of Array Except Self

Given an array of n integers where $n > 1$, `nums`, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Solve it **without division** and in O(n).

For example, given `[1,2,3,4]`, return `[24,12,8,6]`.

Follow up:
Could you solve it with constant space complexity? (Note: The output array **does not** count as extra space for the purpose of space complexity analysis.)

Solution 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

Solution 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

Solution 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

From Leetcoder.

Sliding Window Maximum

Given an array *nums*, there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position.

For example,
Given *nums* = `[1,3,-1,-3,5,3,6,7]`, and *k* = 3.

```
 Window position                Max
 ---------------               -----
 [1  3  -1] -3  5  3  6  7       3
  1 [3  -1  -3] 5  3  6  7       3
  1  3 [-1  -3  5] 3  6  7       5
  1  3  -1 [-3  5  3] 6  7       5
  1  3  -1  -3 [5  3  6] 7       6
  1  3  -1  -3  5 [3  6  7]      7
```

Therefore, return the max sliding window as `[3,3,5,5,6,7]`.

Note:
You may assume *k* is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.

Follow up:
Could you solve it in linear time?

1. How about using a data structure such as deque (double-ended queue)?
2. The queue size need not be the same as the window's size.
3. Remove redundant elements and the queue should store only elements that need to be considered.

Solution 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]

```java
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

Solution 2

The data structure used is know as Monotonic Queue. Click here for more information.

You can also view more solution on Github

```
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

Solution 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

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

For example,

Consider the following matrix:

```
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

Given **target** = `5`, return `true`.

Given **target** = `20`, return `false`.

Solution 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

Solution 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

Solution 3

```
public class Solution {
public boolean searchMatrix(int[][] matrix, int target) {
    int m=matrix.length, n=matrix[0].length, i=0, j=n-1;
    while (i<m && j>=0) {
        if (matrix[i][j]==target) return true;
        else if (matrix[i][j]<target) i++;
        else j--;
    }
    return false;
}
```

}

written by simkieu original link here

From Leetcoder.

Different Ways to Add Parentheses

Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are `+`, `-` and `*`.

Example 1
Input: `"2-1-1"`.

```
((2-1)-1) = 0
(2-(1-1)) = 2
```

Output: `[0, 2]`

Example 2
Input: `"2*3-4*5"`

```
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10
```

Output: `[-34, -14, -10, -10, 10]`

Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Valid Anagram

Given two strings *s* and *t*, write a function to determine if *t* is an anagram of *s*.

For example,
s = "anagram", *t* = "nagaram", return true.
s = "rat", *t* = "car", return false.

Note:
You may assume the string contains only lowercase alphabets.

Follow up:
What if the inputs contain unicode characters? How would you adapt your solution to such case?

Solution 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

Solution 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

Solution 3

Hash Table

This idea uses a hash table to record the times of appearances of each letter in the two strings `s` and `t`. For each letter in `s`, it increases the counter by `1` while for each letter in `t`, it decreases the counter by `1`. Finally, all the counters will be `0` if they two are anagrams of each other.

The first implementation uses the built-in `unordered_map` and takes 36 ms.

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        if (s.length() != t.length()) return false;
        int n = s.length();
        unordered_map<char, int> counts;
        for (int i = 0; i < n; i++) {
            counts[s[i]]++;
            counts[t[i]]--;
        }
        for (auto count : counts)
            if (count.second) return false;
        return true;
    }
};
```

Since the problem statement says that "the string contains only lowercase alphabets", we can simply use an array to simulate the `unordered_map` and speed up the code. The following implementation takes 12 ms.

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        if (s.length() != t.length()) return false;
        int n = s.length();
        int counts[26] = {0};
        for (int i = 0; i < n; i++) {
            counts[s[i] - 'a']++;
            counts[t[i] - 'a']--;
        }
        for (int i = 0; i < 26; i++)
            if (counts[i]) return false;
        return true;
    }
};
```

Sorting

For two anagrams, once they are sorted in a fixed order, they will become the same.

This code is much shorter (this idea can be done in just 1 line using Python as here). However, it takes much longer time --- 76 ms in C++.

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        sort(s.begin(), s.end());
        sort(t.begin(), t.end());
        return s == t;
    }
};
```

written by jianchao.li.fighter original link here

From Leetcoder.

Shortest Word Distance

Given a list of words and two words*word1* and *word2*, return the shortest distance between these two words in the list.

For example,
Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`.

Given *word1* = `“coding”`, *word2* = `“practice”`, return 3.
Given *word1* = `"makes"`, *word2* = `"coding"`, return 1.

Note:
You may assume that *word1* **does not equal to** *word2*, and *word1* and *word2* are both in the list.

Solution 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

Solution 2

```
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

Solution 3

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

From Leetcoder.

Shortest Word Distance II

This is a **follow up** of Shortest Word Distance. The only difference is now you are given the list of words and your method will be called *repeatedly* many times with different parameters. How would you optimize it?

Design a class which receives a list of words in the constructor, and implements a method that takes two words *word1* and *word2* and return the shortest distance between these two words in the list.

For example,
Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`.

Given *word1* = `“coding”`, *word2* = `“practice”`, return 3.
Given *word1* = `"makes"`, *word2* = `"coding"`, return 1.

Note:
You may assume that *word1* **does not equal to** *word2*, and *word1* and *word2* are both in the list.

Solution 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

Solution 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

Solution 3

Special thanks to @qianzhige and @jeantimex

The basic idea is using HashMap to store the position of each word in the array in constructor function. Then in 'shortest' function, use the method similar as merge sort to get the minimum difference between indices of word1 and word2.

Time complexity of 'shortest' function = O(NumberOfWord1 + NumberOfWord2)

Extra space = O(NumberOfWords)

```
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++) {
            if (!map.containsKey(words[i]))
                map.put(words[i], new LinkedList<Integer>());
            map.get(words[i]).add(i);
        }
    }
    public int shortest(String word1, String word2) {
        List<Integer> list1 = map.get(word1);
        List<Integer> list2 = map.get(word2);
        int i = 0, j = 0, res = Integer.MAX_VALUE;
        while (i < list1.size() && j < list2.size()) {
            int index1 = list1.get(i);
            int index2 = list2.get(j);
            if (index1 < index2) {
                i++;
                res = Math.min(res, index2 - index1);
            }
            else {
                j++;
                res = Math.min(res, index1 - index2);
            }
        }
        return res;
    }
}
```

written by Pixel_ original link here

Shortest Word Distance III

This is a **follow up** of Shortest Word Distance. The only difference is now *word1* could be the same as *word2*.

Given a list of words and two words *word1* and *word2*, return the shortest distance between these two words in the list.

word1 and *word2* may be the same and they represent two individual words in the list.

For example,
Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`.

Given *word1* = `“makes”`, *word2* = `“coding”`, return 1.
Given *word1* = `"makes"`, *word2* = `"makes"`, return 3.

Note:
You may assume *word1* and *word2* are both in the list.

Solution 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

Solution 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

Solution 3

Shortest Word Distance I:

```
public class Solution {
    public int shortestWordDistance(String[] words, String word1, String word2) {
        int index = -1;
        int min = words.length;
        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])) {
                    min = Math.min(i - index, min);
                }
                index = i;
            }
        }
        return min;
    }
}
```

Shortest Word Distance III:

```
public class Solution {
        public int shortestWordDistance(String[] words, String word1, String word
2) {
            int index = -1;
            int min = words.length;
            for (int i = 0; i < words.length; i++) {
                if (words[i].equals(word1) || words[i].equals(word2)) {
                    if (index != -1 && (word1.equals(word2) || !words[index].equa
ls(words[i]))) {
                        min = Math.min(i - index, min);
                    }
                    index = i;
                }
            }
            return min;
        }
    }
```

written by blackminimi original link here

Strobogrammatic Number

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to determine if a number is strobogrammatic. The number is represented as a string.

For example, the numbers "69", "88", and "818" are all strobogrammatic.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Strobogrammatic Number II

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Find all strobogrammatic numbers that are of length = n.

For example,
Given n = 2, return `["11","69","88","96"]`.

1. Try to use recursion and notice that it should recurse with n - 2 instead of n - 1.

Solution 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

Solution 2

```
public class Solution {
    public List<String> findStrobogrammatic(int n) {
        List<String> one = Arrays.asList("0", "1", "8"), two = Arrays.asList(""),
r = two;
        if(n%2 == 1)
            r = one;
        for(int i=(n%2)+2; i<=n; i+=2){
            List<String> newList = new ArrayList<>();
            for(String str : r){
                if(i != n)
                    newList.add("0" + str + "0");
                newList.add("1" + str + "1");
                newList.add("6" + str + "9");
                newList.add("8" + str + "8");
                newList.add("9" + str + "6");
            }
            r = newList;
        }
        return r;
    }

}
```

written by symgates original link here

Solution 3

```
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

From Leetcoder.

Strobogrammatic Number III

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to count the total strobogrammatic numbers that exist in the range of low

For example,
Given low = "50", high = "100", return 3. Because 69, 88, and 96 are three strobogrammatic numbers.

Note:
Because the range might be a large number, the *low* and *high* numbers are represented as string.

Solution 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

Solution 2

Construct char array from `lenLow` to `lenHigh` and increase `count` when `s` is between `low` and `high`. Add the stro pairs from outside to inside until `left > right`.

```
char[][] pairs = {{'0', '0'}, {'1', '1'}, {'6', '9'}, {'8', '8'}, {'9', '6'}};
int count = 0;

public int strobogrammaticInRange(String low, String high) {
    for(int len = low.length(); len <= high.length(); len++) {
        dfs(low, high, new char[len], 0, len - 1);
    }
    return count;
}

public void dfs(String low, String high, char[] c, int left, int right) {
    if(left > right) {
        String s = new String(c);
        if((s.length() == low.length() && s.compareTo(low) < 0) ||
           (s.length() == high.length() && s.compareTo(high) > 0)) return;
        count++;
        return;
    }

    for(char[] p : pairs) {
        c[left] = p[0];
        c[right] = p[1];
        if(c.length != 1 && c[0] == '0') continue;
        if(left < right || left == right && p[0] == p[1]) dfs(low, high, c, left
+ 1, right - 1);
    }
}
```

written by yavinci original link here

Solution 3

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

Group Shifted Strings

Given a string, we can "shift" each of its letter to its successive letter, for example: `"abc" -> "bcd"`. We can keep "shifting" which forms the sequence:

```
"abc" -> "bcd" -> ... -> "xyz"
```

Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.

For example, given: `["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"]`, Return:

```
[
  ["abc","bcd","xyz"],
  ["az","ba"],
  ["acef"],
  ["a","z"]
]
```

Note: For the return value, each *inner* list's elements must follow the lexicographic order.

Solution 1

```java
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

Solution 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

Solution 3

Explanation

The basic idea is to set a key for each group: the sum of the difference between the adjacent chars in one string. Then we can easily group the strings belonging to the same shifting sequence with the same key. The code is as the following:

```
public List<List<String>> groupStrings(String[] strs) {
    HashMap<String, ArrayList<String>> map = new HashMap<String, ArrayList<String
>>();
    Arrays.sort(strs);
    for (String s : strs) {
        String key = "";
        for (int i = 1; i < s.length(); i++)
            key += String.format("%2d", (s.charAt(i) - s.charAt(i-1) + 26) % 26);
//Difference from the previous char.
        if (!map.containsKey(key)) map.put(key, new ArrayList<String>());
        map.get(key).add(s);
    }
    return new ArrayList<List<String>>(map.values());
}
```

written by Pixel_ original link here

From Leetcoder.

Count Univalue Subtrees

Given a binary tree, count the number of uni-value subtrees.

A Uni-value subtree means all nodes of the subtree have the same value.

For example:
Given binary tree,

```
              5
             / \
            1   5
           / \   \
          5   5   5
```

return `4`.

Solution 1

```java
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

Solution 2

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

Solution 3

```
public class Solution {
    public int countUnivalSubtrees(TreeNode root) {
        int[] arr = new int[1];
        postOrder(arr, root);
        return arr[0];
    }
    public boolean postOrder (int[] arr, TreeNode node) {
        if (node == null) return true;
        boolean left = postOrder(arr, node.left);
        boolean right = postOrder(arr, node.right);
        if (left && right) {
            if (node.left != null && node.left.val != node.val) return false;
            if (node.right != null && node.right.val != node.val) return false;
            arr[0]++;
            return true;
        }
        return false;
    }
}
```

written by saybia1993 original link here

From Leetcoder.

Flatten 2D Vector

Implement an iterator to flatten a 2d vector.

For example,
Given 2d vector =

```
[
  [1,2],
  [3],
  [4,5,6]
]
```

By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: `[1,2,3,4,5,6]`.

1. How many variables do you need to keep track?
2. Two variables is all you need. Try with `x` and `y`.
3. Beware of empty rows. It could be the first few rows.
4. To write correct code, think about the invariant to maintain. What is it?
5. The invariant is `x` and `y` must always point to a valid point in the 2d vector. Should you maintain your invariant *ahead of time* or *right when you need it*?
6. Not sure? Think about how you would implement `hasNext()`. Which is more complex?
7. Common logic in two different places should be refactored into a common method.

Follow up:
As an added challenge, try to code it using only iterators in C++ or iterators in Java.

Solution 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

Solution 2

//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

Solution 3

- 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

From Leetcoder.

Meeting Rooms

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...]` (s_i i), determine if a person could attend all meetings.

For example,
Given `[[0, 30],[5, 10],[15, 20]]`,
return `false`.

Solution 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

Solution 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

Solution 3

```
public boolean canAttendMeetings(Interval[] intervals) {
        int len=intervals.length;
        if(len==0){
            return true;
        }
        int[]begin=new int[len];
        int[]stop=new int[len];
        for(int i=0;i<len;i++){
            begin[i]=intervals[i].start;
            stop[i]=intervals[i].end;
        }
        Arrays.sort(begin);
        Arrays.sort(stop);
        int endT=0;
        for(int i=1;i<len;i++){
            if(begin[i]<stop[i-1]){
                return false;
            }
        }
        return true;
}
```

written by printf_ll_ original link here

From Leetcoder.

Meeting Rooms II

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...]` (s_i i), find the minimum number of conference rooms required.

For example,
Given `[[0, 30],[5, 10],[15, 20]]`,
return `2`.

Solution 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

Solution 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

Solution 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

Numbers can be regarded as product of its factors. For example,

```
8 = 2 x 2 x 2;
  = 2 x 4.
```

Write a function that takes an integer n and return all possible combinations of its factors.

Note:

1. Each combination's factors must be sorted ascending, for example: The factors of 2 and 6 is `[2, 6]`, not `[6, 2]`.
2. You may assume that n is always positive.
3. Factors should be greater than 1 and less than n.

Examples:
input: `1`
output:

```
[]
```

input: `37`
output:

```
[]
```

input: `12`
output:

```
[
  [2, 6],
  [2, 2, 3],
  [3, 4]
]
```

input: `32`
output:

```
[
  [2, 16],
  [2, 2, 8],
  [2, 2, 2, 4],
  [2, 2, 2, 2, 2],
  [2, 4, 4],
  [4, 8]
]
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Verify Preorder Sequence in Binary Search Tree

Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree.

You may assume each number in the sequence is unique.

Follow up:
Could you do it using only constant space complexity?

Solution 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.

```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

Solution 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

Solution 3

A BST's left child is always < root and right child is always > root.

```
public boolean verifyPreorder(int[] preorder) {
    if(preorder == null || preorder.length == 0) return true;
    return verify(preorder, 0, preorder.length - 1);
}

private boolean verify(int[] a, int start, int end) {
    if(start >= end) return true;
    int pivot = a[start];
    int bigger = -1;
    for(int i = start + 1; i <= end; i++) {
        if(bigger == -1 && a[i] > pivot) bigger = i;
        if(bigger != -1 && a[i] < pivot) return false;
    }
    if(bigger == -1) {
        return verify(a, start + 1, end);
    } else {
        return verify(a, start + 1, bigger - 1) && verify(a, bigger, end);
    }
}
```

written by qianzhige original link here

From Leetcoder.

Paint House

There are a row of *n* houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a `n x 3` cost matrix. For example, `costs[0][0]` is the cost of painting house 0 with color red; `costs[1][2]` is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Note:
All costs are positive integers.

Solution 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 ya09208 original link here

Solution 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

Solution 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

From Leetcoder.

Binary Tree Paths

Given a binary tree, return all root-to-leaf paths.

For example, given the following binary tree:

```
   1
 /   \
2     3
 \
  5
```

All root-to-leaf paths are:

```
["1->2->5", "1->3"]
```

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Add Digits

Given a non-negative integer `num`, repeatedly add all its digits until the result has only one digit.

For example:

Given `num = 38`, the process is like: `3 + 8 = 11`, `1 + 1 = 2`. Since `2` has only one digit, return it.

Follow up:
Could you do it without any loop/recursion in O(1) runtime?

1. A naive implementation of the above process is trivial. Could you come up with other methods?
2. What are all the possible results?
3. How do they occur, periodically or randomly?
4. You may find this Wikipedia article useful.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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:

- $\mathrm{dr}(n) = 0$ if $n == 0$
- $\mathrm{dr}(n) = (b\text{-}1)$ if $n \mathrel{!=} 0$ and $n \% (b\text{-}1) == 0$
- $\mathrm{dr}(n) = n \bmod (b\text{-}1)$ if $n \% (b\text{-}1) \mathrel{!=} 0$

or

- $\mathrm{dr}(n) = 1 + (n - 1) \% 9$

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 $(b\text{-}1)$.

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

Solution 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 ≡ 1 (mod 9), and thus a*n*10^*n* + ... + *a*1*10 + a_0 ≡ 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

Solution 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

From Leetcoder.

3Sum Smaller

Given an array of n integers *nums* and a *target*, find the number of index triplets `i, j, k` with `0 that satisfy the condition nums[i] + nums[j] + nums[k] .`

For example, given *nums* = `[-2, 0, 1, 3]`, and *target* = 2.

Return 2. Because there are two triplets which sums are less than 2:

```
[-2, 0, 1]
[-2, 0, 3]
```

Follow up:
Could you solve it in $O(n^2)$ runtime?

Solution 1

```
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

Solution 2

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

Solution 3

We sort the array first. Then, for each element, we use the two pointer approach to find the number of triplets that meet the requirements.

Let me illustrate how the two pointer technique works with an example:

```
target = 2

  i  lo    hi
[-2, 0, 1, 3]
```

We use a for loop (index i) to iterate through each element of the array. For each i, we create two pointers, lo and hi, where lo is initialized as the next element of i, and hi is initialized at the end of the array. If we know that nums[i] + nums[lo] + nums[hi] < target, then we know that since the array is sorted, we can replace hi with any element from lo+1 to nums.length-1, and the requirements will still be met. Just like in the example above, we know that since -2 + 0 + 3 < 2, we can replace hi (3) with 1, and it would still work. Therefore, we can just add hi - lo to the triplet count.

```
public class Solution {
    public int threeSumSmaller(int[] nums, int target) {
        int result = 0;
        Arrays.sort(nums);
        for(int i = 0; i <= nums.length-3; i++) {
            int lo = i+1;
            int hi = nums.length-1;
            while(lo < hi) {
                if(nums[i] + nums[lo] + nums[hi] < target) {
                    result += hi - lo;
                    lo++;
                } else {
                    hi--;
                }
            }
        }
        return result;
    }
}
```

written by kevinhsu original link here

Single Number III

Given an array of numbers `nums`, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

For example:

Given `nums = [1, 2, 1, 3, 2, 5]`, return `[3, 5]`.

Note:

1. The order of the result is not important. So in the above example, `[5, 3]` is also correct.
2. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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++:

```
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

Solution 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.

Let’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

Solution 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

From Leetcoder.

Graph Valid Tree

Given `n` nodes labeled from `0` to `n - 1` and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:

Given `n = 5` and `edges = [[0, 1], [0, 2], [0, 3], [1, 4]]`, return `true`.

Given `n = 5` and `edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]]`, return `false`.

1. Given `n = 5` and `edges = [[0, 1], [1, 2], [3, 4]]`, what should your return? Is this case a valid tree?
2. According to the definition of tree on Wikipedia: "a tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree."

Note: you can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]` and thus will not appear together in `edges`.

Solution 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

Solution 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

Solution 3

```
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

Write a program to check whether a given number is an ugly number.

Ugly numbers are positive numbers whose prime factors only include `2, 3, 5`. For example, `6, 8` are ugly while `14` is not ugly since it includes another prime factor `7`.

Note that `1` is typically treated as an ugly number.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Ugly Number II

Write a program to find the `n`-th ugly number.

Ugly numbers are positive numbers whose prime factors only include `2, 3, 5`. For example, `1, 2, 3, 4, 5, 6, 8, 9, 10, 12` is the sequence of the first `10` ugly numbers.

Note that `1` is typically treated as an ugly number.

1. The naive approach is to call `isUgly` for every number until you reach the nth one. Most numbers are *not* ugly. Try to focus your effort on generating only the ugly ones.
2. An ugly number must be multiplied by either 2, 3, or 5 from a smaller ugly number.
3. The key is how to maintain the order of the ugly numbers. Try a similar approach of merging from three sorted lists: L_1, L_2, and L_3.
4. Assume you have U_k, the kth ugly number. Then U_{k+1} must be Min(L_1 * 2, L_2 * 3, L_3 * 5).

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 2

```
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

Solution 3

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, ... 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) 1×2, 2×2, 3×2, 4×2, 5×2, …
(2) 1×3, 2×3, 3×3, 4×3, 5×3, …
(3) 1×5, 2×5, 3×5, 4×5, 5×5, …
```

We can find that every subsequence is the ugly-sequence itself (1, 2, 3, 4, 5, ...) 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

```java
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

From Leetcoder.

Paint House II

There are a row of n houses, each house can be painted with one of the$k$ colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a `n x k` cost matrix. For example, `costs[0][0]` is the cost of painting house 0 with color 0; `costs[1][2]` is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

Note:
All costs are positive integers.

Follow up:
Could you solve it in $O(nk)$ runtime?

Solution 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

Solution 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

Solution 3

Explanation: dp[i][j] represents the min paint cost from house 0 to house i when house i use color j; The formula will be dp[i][j] = Math.min(any k!= j| dp[i-1][k]) + costs[i][j].

Take a closer look at the formula, we don't need an array to represent dp[i][j], we only need to know the min cost to the previous house of any color and if the color j is used on previous house to get prev min cost, use the second min cost that are not using color j on the previous house. So I have three variable to record: prevMin, prevMinColor, prevSecondMin. and the above formula will be translated into: dp[currentHouse][currentColor] = (currentColor == prevMinColor? prevSecondMin: prevMin) + costs[currentHouse][currentColor].

```
public class Solution {
public int minCostII(int[][] costs) {
    if(costs == null || costs.length == 0 || costs[0].length == 0) return 0;

    int n = costs.length, k = costs[0].length;
    if(k == 1) return (n==1? costs[0][0] : -1);

    int prevMin = 0, prevMinInd = -1, prevSecMin = 0;//prevSecMin always >= prevM
in
    for(int i = 0; i<n; i++) {
        int min = Integer.MAX_VALUE, minInd = -1, secMin = Integer.MAX_VALUE;
        for(int j = 0; j<k;  j++) {
            int val = costs[i][j] + (j == prevMinInd? prevSecMin : prevMin);
            if(minInd< 0) {min = val; minInd = j;}//when min isn't initialized
            else if(val < min) {//when val < min,
                secMin = min;
                min = val;
                minInd = j;
            } else if(val < secMin) { //when min<=val< secMin
                secMin = val;
            }
        }
        prevMin = min;
        prevMinInd = minInd;
        prevSecMin = secMin;
    }
    return prevMin;
}
}
```

written by Tracy123 original link here

Palindrome Permutation

Given a string, determine if a permutation of the string could form a palindrome.

For example,
`"code"` -> False, `"aab"` -> True, `"carerac"` -> True.

1. Consider the palindromes of odd vs even length. What difference do you notice?
2. Count the frequency of each character.
3. If each character occurs even number of times, then it must be a palindrome. How about character which occurs odd number of times?

Solution 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

Solution 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.

```java
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

Solution 3

Explanation

The basic idea is using HashSet to find the number of single characters, which should be at most 1.

```
public boolean canPermutePalindrome(String s) {
    Set<Character>set = new HashSet<Character>();
    for (char c : s.toCharArray())
        if (set.contains(c)) set.remove(c);// If char already exists in set, then
remove it from set
        else set.add(c);// If char doesn't exists in set, then add it to set
    return set.size() <= 1;
}
```

written by Pixel_ original link here

From Leetcoder.

Palindrome Permutation II

Given a string `s`, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.

For example:

Given `s = "aabb"`, return `["abba", "baab"]`.

Given `s = "abc"`, return `[]`.

1. If a palindromic permutation exists, we just need to generate the first half of the string.
2. To generate all distinct permutations of a (half of) string, use a similar approach from: Permutations II or Next Permutation.

Solution 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

Solution 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

Solution 3

We only need to generate the first part of palindrome string, and the remaining part will be a middle character with the reverse of first part.

```
private List<String> list = new ArrayList<>();

public List<String> generatePalindromes(String s) {
    int numOdds = 0; // How many characters that have odd number of count
    int[] map = new int[256]; // Map from character to its frequency
    for (char c: s.toCharArray()) {
        map[c]++;
        numOdds = (map[c] & 1) == 1 ? numOdds+1 : numOdds-1;
    }
    if (numOdds > 1)   return list;

    String mid = "";
    int length = 0;
    for (int i = 0; i < 256; i++) {
        if (map[i] > 0) {
            if ((map[i] & 1) == 1) { // Char with odd count will be in the middle
                mid = "" + (char)i;
                map[i]--;
            }
            map[i] /= 2; // Cut in half since we only generate half string
            length += map[i]; // The length of half string
        }
    }
    generatePalindromesHelper(map, length, "", mid);
    return list;
}
private void generatePalindromesHelper(int[] map, int length, String s, String mi
d) {
    if (s.length() == length) {
        StringBuilder reverse = new StringBuilder(s).reverse(); // Second half
        list.add(s + mid + reverse);
        return;
    }
    for (int i = 0; i < 256; i++) { // backtracking just like permutation
        if (map[i] > 0) {
            map[i]--;
            generatePalindromesHelper(map, length, s + (char)i, mid);
            map[i]++;
        }
    }
}
```

written by lianngg original link here

From Leetcoder.

Missing Number

Given an array containing *n* distinct numbers taken from `0, 1, 2, ..., n`, find the one that is missing from the array.

For example,
Given *nums* = `[0, 1, 3]` return `2`.

Note:
Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

1.XOR

```
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

```
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

```
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

From Leetcoder.

Alien Dictionary

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of words from the dictionary, where **words are sorted lexicographically by the rules of this new language**. Derive the order of letters in this language.

For example,
Given the following words in dictionary,

```
[
  "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"
]
```

The correct order is: `"wertf"`.

Note:

1. You may assume all letters are in lowercase.
2. If the order is invalid, return an empty string.
3. There may be multiple valid order of letters, return any one of them is fine.

Solution 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/predecessor sets. 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(set), 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].add(a)
                break
    chars = set(''.join(words))
    free = chars - set(pre)
    order = ''
    while free:
        a = free.pop()
        order += a
        for b in suc[a]:
            pre[b].discard(a)
            if not pre[b]:
                free.add(b)
    return order * (set(order) == chars)
```

C++ version of solution 2

```
string alienOrder(vector<string>& words) {
    map<char, set<char>> suc, 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].insert(a);
                break;
            }
        }
        s = t;
    }
    set<char> free = chars;
    for (auto p : pre)
        free.erase(p.first);
    string order;
    while (free.size()) {
        char a = *begin(free);
        free.erase(a);
        order += a;
        for (char b : suc[a]) {
            pre[b].erase(a);
            if (pre[b].empty())
                free.insert(b);
        }
    }
    return order.size() == chars.size() ? order : "";
}
```

written by StefanPochmann original link here

Solution 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

Solution 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

From Leetcoder.

Closest Binary Search Tree Value

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note:

- Given target value is a floating point.
- You are guaranteed to have only one unique value in the BST that is closest to the target.

Solution 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

```
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++

```
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

```
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

```
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

```
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:

```
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++

```
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

Solution 2

```
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

Solution 3

```
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

From Leetcoder.

Encode and Decode Strings

Design an algorithm to encode **a list of strings** to **a string**. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:

```
string encode(vector<string> strs) {
  // ... your code
  return encoded_string;
}
```

Machine 2 (receiver) has the function:

```
vector<string> decode(string s) {
  //... your code
  return strs;
}
```

So Machine 1 does:

```
string encoded_string = encode(strs);
```

and Machine 2 does:

```
vector<string> strs2 = decode(encoded_string);
```

`strs2` in Machine 2 should be the same as `strs` in Machine 1.

Implement the `encode` and `decode` methods.

Note:

- The string may contain any possible characters out of 256 valid ascii characters. Your algorithm should be generalized enough to work on any possible characters.
- Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.
- Do not rely on any library method such as `eval` or serialize methods. You should implement your own encode/decode algorithm.

Solution 1

```java
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

Solution 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

Solution 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

From Leetcoder.

Closest Binary Search Tree Value II

Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.

Note:

- Given target value is a floating point.
- You may assume k is always valid, that is: $k \leq$ total nodes.
- You are guaranteed to have only one unique set of k values in the BST that are closest to the target.

Follow up:
Assume that the BST is balanced, could you solve it in less than $O(n)$ runtime (where n = total nodes)?

1. Consider implement these two helper functions:
   i. `getPredecessor(N)`, which returns the next smaller node to N.
   ii. `getSuccessor(N)`, which returns the next larger node to N.
2. Try to assume that each node has a parent pointer, it makes the problem much easier.
3. Without parent pointer we just need to keep track of the path from the root to the current node using a stack.
4. You would need two stacks to track the path in finding predecessor and successor node separately.

Solution 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

Solution 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

Solution 3

```
public class Solution {
    public List<Integer> closestKValues(TreeNode root, double target, int k) {
        LinkedList<Integer> list = new LinkedList<Integer>();
        closestKValuesHelper(list, root, target, k);
        return list;
    }

    /**
     * @return <code>true</code> if result is already found.
     */
    private boolean closestKValuesHelper(LinkedList<Integer> list, TreeNode root,
double target, int k) {
        if (root == null) {
            return false;
        }

        if (closestKValuesHelper(list, root.left, target, k)) {
            return true;
        }

        if (list.size() == k) {
            if (Math.abs(list.getFirst() - target) < Math.abs(root.val - target))
{
                return true;
            } else {
                list.removeFirst();
            }
        }

        list.addLast(root.val);
        return closestKValuesHelper(list, root.right, target, k);
    }
}
```

written by magma917 original link here

From Leetcoder.

Integer to English Words

Convert a non-negative integer to its english words representation. Given input is guaranteed to be less than $2^{31} - 1$.

For example,

```
123 -> "One Hundred Twenty Three"
12345 -> "Twelve Thousand Three Hundred Forty Five"
1234567 -> "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"
```

1. Did you see a pattern in dividing the number into chunk of words? For example, 123 and 123000.
2. Group the number by thousands (3 digits). You can write a helper function that takes a number less than 1000 and convert just that chunk to words.
3. There are many edge cases. What are some good test cases? Does your code work with input such as 0? Or 1000010? (middle chunk is zero and should not be printed out)

Solution 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

Solution 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

Solution 3

```
public class Solution {
    private final String[] belowTen = new String[] {"", "One", "Two", "Three", "F
our", "Five", "Six", "Seven", "Eight", "Nine"};
    private final String[] belowTwenty = new String[] {"Ten", "Eleven", "Twelve",
"Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"
};
    private final String[] belowHundred = new String[] {"", "Ten", "Twenty", "Thi
rty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};

    public String numberToWords(int num) {
        if (num == 0) return "Zero";
        return helper(num);
    }

    private String helper(int num) {
        String result = new String();
        if (num < 10) result = belowTen[num];
        else if (num < 20) result = belowTwenty[num -10];
        else if (num < 100) result = belowHundred[num/10] + " " + helper(num % 10
);
        else if (num < 1000) result = helper(num/100) + " Hundred " +  helper(num
% 100);
        else if (num < 1000000) result = helper(num/1000) + " Thousand " +  helpe
r(num % 1000);
        else if (num < 1000000000) result = helper(num/1000000) + " Million " +
helper(num % 1000000);
        else result = helper(num/1000000000) + " Billion " + helper(num % 1000000
000);
        return result.trim();
    }
}
```

written by iuri.srb original link here

From Leetcoder.

H-Index

Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

According to the definition of h-index on Wikipedia: "A scientist has index h if h of his/her N papers have **at least** h citations each, and the other $N - h$ papers have **no more than** h citations each."

For example, given `citations = [3, 0, 6, 1, 5]`, which means the researcher has `5` papers in total and each of them had received `3, 0, 6, 1, 5` citations respectively. Since the researcher has `3` papers with **at least** `3` citations each and the remaining two with **no more than** `3` citations each, his h-index is `3`.

Note: If there are several possible values for `h`, the maximum one is taken as the h-index.

1. An easy approach is to sort the array first.
2. What are the possible values of h-index?
3. A faster approach is to use extra space.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 1

```java
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

Solution 2

```
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

Solution 3

```
 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

From Leetcoder.

H-Index II

Follow up for H-Index: What if the `citations` array is sorted in ascending order? Could you optimize your algorithm?

1. Expected runtime complexity is in $O(\log n)$ and the input is sorted.

Solution 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

Solution 2

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

Solution 3

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

From Leetcoder.

Paint Fence

There is a fence with n posts, each post can be painted with one of the k colors.

You have to paint all the posts such that no more than two adjacent fence posts have the same color.

Return the total number of ways you can paint the fence.

Note:
n and k are non-negative integers.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Find the Celebrity

Suppose you are at a party with `n` people (labeled from `0` to `n - 1`) and among them, there may exist one celebrity. The definition of a celebrity is that all the other `n - 1` people know him/her but he/she does not know any of them.

Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask questions like: "Hi, A. Do you know B?" to get information of whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).

You are given a helper function `bool knows(a, b)` which tells you whether A knows B. Implement a function `int findCelebrity(n)`, your function should minimize the number of calls to `knows`.

Note: There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party. If there is no celebrity, return `-1`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

First Bad Version

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have `n` versions `[1, 2, ..., n]` and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API `bool isBadVersion(version)` which will return whether `version` is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Perfect Squares

Given a positive integer n, find the least number of perfect square numbers (for example, `1, 4, 9, 16, ...`) which sum to n.

For example, given n = `12`, return `3` because `12 = 4 + 4 + 4`; given n = `13`, return `2` because `13 = 4 + 9`.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

```
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 and only 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

Solution 2

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:

```java
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

Solution 3

These solutions use some number theory (see explanation further down).

Ruby solution

```ruby
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.

```
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

From Leetcoder.

Wiggle Sort

Given an unsorted array `nums`, reorder it **in-place** such that `nums[0] = nums[2]`.

For example, given `nums = [3, 5, 2, 1, 6, 4]`, one possible answer is `[1, 6, 2, 5, 3, 4]`.

Solution 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

Solution 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

Solution 3

Python

```python
class Solution(object):
    def wiggleSort(self, nums):
        for i in xrange(1, len(nums)):
            if (i % 2) ^ (nums[i] > nums[i - 1]):
                nums[i], nums[i - 1] = nums[i - 1], nums[i]


# 125 / 125 test cases passed.
# Status: Accepted
# Runtime: 80 ms
```

Java

```java
public void wiggleSort(int[] nums) {
    for (int i = 1; i < nums.length; ++i) {
        if ((i % 2 == 1) != (nums[i] > nums[i - 1])) {
            int cache = nums[i];
            nums[i] = nums[i-1];
            nums[i-1] = cache;
        }
    }
}

// 125 / 125 test cases passed.
// Status: Accepted
// Runtime: 1 ms
```

written by dietpepsi original link here

From Leetcoder.

Zigzag Iterator

Given two 1d vectors, implement an iterator to return their elements alternately.

For example, given two 1d vectors:

```
v1 = [1, 2]
v2 = [3, 4, 5, 6]
```

By calling *next* repeatedly until *hasNext* returns `false`, the order of elements returned by *next* should be: `[1, 3, 2, 4, 5, 6]`.

Follow up: What if you are given `k` 1d vectors? How well can your code be extended to such cases?

Clarification for the follow up question - Update (2015-09-18):
The "Zigzag" order is not clearly defined and is ambiguous for `k > 2` cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic". For example, given the following input:

```
[1,2,3]
[4,5,6,7]
[8,9]
```

It should return `[1,4,8,2,5,9,3,6,7]`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Expression Add Operators

Given a string that contains only digits `0-9` and a target value, return all possibilities to add **binary** operators (not unary) `+`, `-`, or `*` between the digits so they evaluate to the target value.

Examples:

```
"123", 6 -> ["1+2+3", "1*2*3"]
"232", 8 -> ["2*3+2", "2+3*2"]
"105", 5 -> ["1*0+5","10-5"]
"00", 0 -> ["0+0", "0-0", "0*0"]
"3456237490", 9191 -> []
```

Credits:
Special thanks to @davidtan1890 for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Move Zeroes

Given an array `nums`, write a function to move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

For example, given `nums = [0, 1, 0, 3, 12]`, after calling your function, `nums` should be `[1, 3, 12, 0, 0]`.

Note:

1. You must do this **in-place** without making a copy of the array.
2. Minimize the total number of operations.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int j = 0;
        // move all the nonzero elements advance
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] != 0) {
                nums[j++] = nums[i];
            }
        }
        for (;j < nums.size(); j++) {
            nums[j] = 0;
        }
    }
};
```

written by hyzhang original link here

From Leetcoder.

Peeking Iterator

Given an Iterator class interface with methods: `next()` and `hasNext()`, design and implement a PeekingIterator that support the `peek()` operation -- it essentially peek() at the element that will be returned by the next call to next().

Here is an example. Assume that the iterator is initialized to the beginning of the list: `[1, 2, 3]`.

Call `next()` gets you 1, the first element in the list.

Now you call `peek()` and it returns 2, the next element. Calling `next()` after that ***still*** return 2.

You call `next()` the final time and it returns 3, the last element. Calling `hasNext()` after that should return false.

1. Think of "looking ahead". You want to cache the next element.
2. Is one variable sufficient? Why or why not?
3. Test your design with call order of `peek()` before `next()` vs `next()` before `peek()`.
4. For a clean implementation, check out Google's guava library source code.

Follow up: How would you extend your design to be generic and work with all types, not just integer?

Credits:
Special thanks to @porker2008 for adding this problem and creating all test cases.

Solution 1

```java
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

Solution 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

Solution 3

```
class PeekingIterator : public Iterator {
 public:
 PeekingIterator(const vector<int>& nums) : Iterator(nums) {
    // Initialize any member here.
    // **DO NOT** save a copy of nums and manipulate it directly.
    // You should only use the Iterator interface methods.

}

// Returns the next element in the iteration without advancing the iterator.
int peek() {
    if(hasNext()){
        Iterator it(*this);
        return it.next();
    }
}

// hasNext() and next() should behave the same as in the Iterator interface.
// Override them if needed.
int next() {
    Iterator::next();
}

bool hasNext() const {
    Iterator::hasNext();
}
```

};

written by leebear original link here

From Leetcoder.

Inorder Successor in BST

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

Note: If the given node has no in-order successor in the tree, return `null`.

Solution 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

Solution 2

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

```
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

```
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

Solution 3

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

From Leetcoder.

Walls and Gates

You are given a *m x n* 2D grid initialized with these three possible values.

1. `-1` - A wall or an obstacle.
2. `0` - A gate.
3. `INF` - Infinity means an empty room. We use the value `$2^{31} - 1$ = 2147483647` to represent `INF` as you may assume that the distance to a gate is less than `2147483647`.

Fill each empty room with the distance to its *nearest* gate. If it is impossible to reach a gate, it should be filled with `INF`.

For example, given the 2D grid:

```
INF  -1  0  INF
INF INF INF  -1
INF  -1 INF  -1
  0  -1 INF INF
```

After running your function, the 2D grid should be:

```
  3  -1   0   1
  2   2   1  -1
  1  -1   2  -1
  0  -1   3   4
```

Solution 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

Solution 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

Solution 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

From Leetcoder.

Find the Duplicate Number

Given an array *nums* containing $n + 1$ integers where each integer is between 1 and n (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

Note:

1. You **must not** modify the array (assume the array is read only).
2. You must use only constant, $O(1)$ extra space.
3. Your runtime complexity should be less than `O(n^2)`.
4. There is only one duplicate number in the array, but it could be repeated more than once.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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!

```java
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

From Leetcoder.

Unique Word Abbreviation

An abbreviation of a word follows the form <first letter><number><last letter>. Below are some examples of word abbreviations:

```
a) it                      --> it    (no abbreviation)

     1
b) d|o|g                   --> d1g

               1    1  1
     1---5----0----5--8
c) i|nternationalizatio|n  --> i18n

               1
     1---5----0
d) l|ocalizatio|n          --> l10n
```

Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no *other* word from the dictionary has the same abbreviation.

Example:

```
Given dictionary = [ "deer", "door", "cake", "card" ]

isUnique("dear") ->
false

isUnique("cart") ->
true

isUnique("cane") ->
false

isUnique("make") ->
true
```

Solution 1

```
public class ValidWordAbbr {
    HashMap<String, String> map;
    public ValidWordAbbr(String[] dictionary) {
        map = new HashMap<String, String>();
        for(String str:dictionary){
            String key = getKey(str);
            // If there is more than one string belong to the same key
            // then the key will be invalid, we set the value to ""
            if(map.containsKey(key)){
                if(!map.get(key).equals(str)){
                    map.put(key, "");
                }
            }
            else{
                map.put(key, str);
            }
        }
    }

    public boolean isUnique(String word) {
        return !map.containsKey(getKey(word))||map.get(getKey(word)).equals(word)
;
    }

    String getKey(String str){
        if(str.length()<=2) return str;
        return str.charAt(0)+Integer.toString(str.length()-2)+str.charAt(str.leng
th()-1);
    }
}
```

written by skylinebaby original link here

Solution 2

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

Solution 3

```
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

From Leetcoder.

Game of Life

According to the Wikipedia's article: "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

Given a *board* with *m* by *n* cells, each cell has an initial state *live* (1) or *dead* (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population..
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.

Follow up:

1. Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.
2. In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 2

To solve it in place, we use 2 bits to store 2 states:

```
[2nd bit, 1st bit] = [next state, current state]

- 00  dead (next) <- dead (current)
- 01  dead (next) <- live (current)
- 10  live (next) <- dead (current)
- 11  live (next) <- live (current)
```

- In the beginning, every cell is either `00` or `01`.
- Notice that `1st` state is independent of `2nd` state.
- Imagine all cells are instantly changing from the `1st` to the `2nd` state, at the same time.
- Let's count # of neighbors from `1st` state and set `2nd` state bit.
- Since every `2nd` state is by default dead, no need to consider transition `01 -> 00`.
- In the end, delete every cell's `1st` state by doing `>> 1`.

For each cell's `1st` bit, check the 8 pixels around itself, and set the cell's `2nd` bit.

- Transition `01 -> 11`: when `board == 1` and `lives >= 2 && lives <= 3`.
- Transition `00 -> 10`: when `board == 0` and `lives == 3`.

To get the current state, simply do

```
board[i][j] & 1
```

To get the next state, simply do

```
board[i][j] >> 1
```

Hope this helps!

```java
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 the 2nd bit will become 1.
            if(board[i][j] == 1 && lives >= 2 && lives <= 3) {
                board[i][j] = 3; // Make the 2nd bit 1: 01 ---> 11
            }
            if(board[i][j] == 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 x = Math.max(i - 1, 0); x <= Math.min(i + 1, m - 1); x++) {
        for(int y = Math.max(j - 1, 0); y <= Math.min(j + 1, n - 1); y++) {
            lives += board[x][y] & 1;
        }
    }
    lives -= board[i][j] & 1;
    return lives;
}
```

written by yavinci original link here

Solution 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

From Leetcoder.

Word Pattern

Given a `pattern` and a string `str`, find if `str` follows the same pattern.

Here **follow** means a full match, such that there is a bijection between a letter in `pattern` and a **non-empty** word in `str`.

Examples:

1. pattern = `"abba"`, str = `"dog cat cat dog"` should return true.
2. pattern = `"abba"`, str = `"dog cat cat fish"` should return false.
3. pattern = `"aaaa"`, str = `"dog cat cat dog"` should return false.
4. pattern = `"abba"`, str = `"dog dog dog dog"` should return false.

Notes:
You may assume `pattern` contains only lowercase letters, and `str` contains lowercase letters separated by a single space.

Credits:
Special thanks to @minglotus6 for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Word Pattern II

Given a `pattern` and a string `str`, find if `str` follows the same pattern.

Here **follow** means a full match, such that there is a bijection between a letter in `pattern` and a **non-empty** substring in `str`.

Examples:

1. pattern = `"abab"`, str = `"redblueredblue"` should return true.
2. pattern = `"aaaa"`, str = `"asdasdasdasd"` should return true.
3. pattern = `"aabb"`, str = `"xyzabcxzyabc"` should return false.

Notes:
You may assume both `pattern` and `str` contains only lowercase letters.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Nim Game

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend.

1. If there are 5 stones in the heap, could you figure out a way to remove the stones such that you will always be the winner?

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

```
class Solution(object):
    def canWinNim(self, n):
        """
        :type n: int
        :rtype: bool
        """
        # strategy: the one with 4 remaining must loose
        # A, B players
        # if n == 4k, then at each round B can make A+B both take 4,
        # eventually leave 4 to A, A lose
        # if n == 4k + i (i <= 3), then A can always take i first and B will
        # finanly lose as he faces above scenario like A

        return bool(n%4!=0)
```

written by pennlio original link here

From Leetcoder.

Flip Game

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: `+` and `-`, you and your friend take turns to flip two **consecutive** `"++"` into `"--"`. The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to compute all possible states of the string after one valid move.

For example, given `s = "++++"`, after one move, it may become one of the following states:

```
[
  "--++",
  "+--+",
  "++--"
]
```

If there is no valid move, return an empty list `[]`.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Flip Game II

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: `+` and `-`, you and your friend take turns to flip two **consecutive** `"++"` into `"--"`. The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to determine if the starting player can guarantee a win.

For example, given `s = "++++"`, return true. The starting player can guarantee a win by flipping the middle `"++"` to become `"+--+"`.

Follow up:
Derive your algorithm's runtime complexity.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Find Median from Data Stream

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Examples:
`[2,3,4]` , the median is `3`

`[2,3]`, the median is `(2 + 3) / 2 = 2.5`

Design a data structure that supports the following two operations:

- void addNum(int num) - Add a integer number from the data stream to the data structure.
- double findMedian() - Return the median of all elements so far.

For example:

```
add(1)
add(2)
findMedian() -> 1.5
add(3)
findMedian() -> 2
```

Credits:
Special thanks to @Louis1992 for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 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

From Leetcoder.

Best Meeting Point

A group of two or more people wants to meet and minimize the total travel distance. You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group. The distance is calculated using Manhattan Distance, where distance(p1, p2) = `|p2.x - p1.x| + |p2.y - p1.y|`.

For example, given three people living at `(0,0)`, `(0,4)`, and `(2,2)`:

```
1 - 0 - 0 - 0 - 1
|   |   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |   |
0 - 0 - 1 - 0 - 0
```

The point `(0,2)` is an ideal meeting point, as the total travel distance of 2+2+2=6 is minimal. So return 6.

1. Try to solve it in one dimension first. How can this solution apply to the two dimension case?

Solution 1

```java
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

Solution 2

As long as you have different numbers of people on your left and on your right, moving a little to the side with more people decreases the sum of distances. So to minimize it, you must have equally many people on your left and on your right. Same with above/below.

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.

```java
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

Solution 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

From Leetcoder.

Serialize and Deserialize Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

For example, you may serialize the following tree

```
    1
   / \
  2   3
     / \
    4   5
```

as `"[1,2,3,null,null,4,5]"`, just the same as how LeetCode OJ serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.
Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

Credits:
Special thanks to @Louis1992 for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

```java
 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

From Leetcoder.

Binary Tree Longest Consecutive Sequence

Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

For example,

```
   1
    \
     3
    / \
   2   4
        \
         5
```

Longest consecutive sequence path is `3-4-5`, so return `3`.

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

Longest consecutive sequence path is `2-3`,not `3-2-1`, so return `2`.

Solution 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

Solution 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

Solution 3

```
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

From Leetcoder.

Bulls and Cows

You are playing the following Bulls and Cows game with your friend: You write down a number and ask your friend to guess what the number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your secret number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number.

For example:

```
Secret number:  "1807"
Friend's guess: "7810"
```

Hint: `1` bull and `3` cows. (The bull is `8`, the cows are `0`, `1` and `7`.)
Write a function to return a hint according to the secret number and friend's guess, use `A` to indicate the bulls and `B` to indicate the cows. In the above example, your function should return `"1A3B"`.

Please note that both secret number and friend's guess may contain duplicate digits, for example:

```
Secret number:  "1123"
Friend's guess: "0111"
```

In this case, the 1st `1` in friend's guess is a bull, the 2nd or 3rd `1` is a cow, and your function should return `"1A1B"`.
You may assume that the secret number and your friend's guess only contain digits, and their lengths are always equal.

Credits:
Special thanks to @jeantimex for adding this problem and creating all test cases.

Solution 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

Solution 2

```
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

Solution 3

The idea is simple, if two char is match, add aCnt, otherwise, record it and process bCnt in second pass.

```
class Solution {
public:
    // only contains digits
    string getHint(string secret, string guess) {
        int aCnt = 0;
        int bCnt = 0;
        vector<int> sVec(10, 0); // 0 ~ 9 for secret
        vector<int> gVec(10, 0); // 0 ~ 9 for guess
        if (secret.size() != guess.size() || secret.empty()) { return "0A0B"; }
        for (int i = 0; i < secret.size(); ++i) {
            char c1 = secret[i]; char c2 = guess[i];
            if (c1 == c2) {
                ++aCnt;
            } else {
                ++sVec[c1-'0'];
                ++gVec[c2-'0'];
            }
        }
        // count b
        for (int i = 0; i < sVec.size(); ++i) {
            bCnt += min(sVec[i], gVec[i]);
        }
        return to_string(aCnt) + 'A' + to_string(bCnt) + 'B';
    }
};
```

written by moumoutsay original link here

From Leetcoder.

Longest Increasing Subsequence

Given an unsorted array of integers, find the length of longest increasing subsequence.

For example,
Given `[10, 9, 2, 5, 3, 7, 101, 18]`,
The longest increasing subsequence is `[2, 3, 7, 101]`, therefore the length is `4`.
Note that there may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in $O(n^2)$ complexity.

Follow up: Could you improve it to $O(n \log n)$ time complexity?

Credits:
Special thanks to @pbrother for adding this problem and creating all test cases.

Solution 1

```java
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

Solution 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

Solution 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

```
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

From Leetcoder.

Remove Invalid Parentheses

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses `(` and `)`.

Examples:

```
"()())()" -> ["()()()", "(())()"]
"(a)())()" -> ["(a)()()", "(a())()"]
")(" -> [""]
```

Credits:
Special thanks to @hpplayer for adding this problem and creating all test cases.

Solution 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;
```

```java
        }

        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

Solution 2

Here I share my `DFS` or `backtracking` solution. It's `10X` faster than optimized `BFS`.

1. Limit max removal `rmL` and `rmR` for backtracking boundary. Otherwise it will exhaust all possible valid substrings, not shortest ones.
2. Scan from left to right, avoiding invalid strs (on the fly) by checking num of `open` parens.
3. If it's `'('`, either use it, or remove it.
4. If it's `'('`, either use it, or remove it.
5. Otherwise just append it.
6. Lastly set `StringBuilder` to the last decision point.

In each step, make sure:

1. `i` does not exceed `s.length()`.
2. Max removal `rmL` `rmR` and num of `open` parens are non negative.
3. De-duplicate by adding to a `HashSet`.

Compared to `106 ms` `BFS (Queue & Set)`, it's faster and easier. Hope it helps! Thanks.

```java
public List<String> removeInvalidParentheses(String s) {
    Set<String> res = new HashSet<>();
    int rmL = 0, rmR = 0;
    for(int i = 0; i < s.length(); i++) {
        if(s.charAt(i) == '(') rmL++;
        if(s.charAt(i) == ')') {
            if(rmL != 0) rmL--;
            else rmR++;
        }
    }
    DFS(res, s, 0, rmL, rmR, 0, new StringBuilder());
    return new ArrayList<String>(res);
}

public void DFS(Set<String> res, String s, int i, int rmL, int rmR, int open, Stri
ngBuilder sb) {
    if(i == s.length() && rmL == 0 && rmR == 0 && open == 0) {
        res.add(sb.toString());
        return;
    }
    if(i == s.length() || rmL < 0 || rmR < 0 || open < 0) return;

    char c = s.charAt(i);
    int len = sb.length();

    if(c == '(') {
        DFS(res, s, i + 1, rmL - 1, rmR, open, sb);
        DFS(res, s, i + 1, rmL, rmR, open + 1, sb.append(c));

    } else if(c == ')') {
        DFS(res, s, i + 1, rmL, rmR - 1, open, sb);
        DFS(res, s, i + 1, rmL, rmR, open - 1, sb.append(c));

    } else {
        DFS(res, s, i + 1, rmL, rmR, open, sb.append(c));
    }

    sb.setLength(len);
}
```

written by yavinci original link here

Solution 3

```
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

From Leetcoder.

Smallest Rectangle Enclosing Black Pixels

An image is represented by a binary matrix with `0` as a white pixel and `1` as a black pixel. The black pixels are connected, i.e., there is only one black region. Pixels are connected horizontally and vertically. Given the location `(x, y)` of one of the black pixels, return the area of the smallest (axis-aligned) rectangle that encloses all black pixels.

For example, given the following image:

```
[
  "0010",
  "0110",
  "0100"
]
```

and `x = 0`, `y = 2`,
Return `6`.

Solution 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++

```
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

```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)

```java
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

Solution 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

Solution 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

From Leetcoder.

Range Sum Query - Immutable

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* ($i \le j$), inclusive.

Example:

```
Given nums = [-2, 0, 3, -5, 2, -1]

sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

Note:

1. You may assume that the array does not change.
2. There are many calls to *sumRange* function.

Solution 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

Solution 2

```
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

Solution 3

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

```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

From Leetcoder.

Range Sum Query 2D - Immutable

Given a 2D matrix *matrix*, find the sum of the elements inside the rectangle defined by its upper left corner (*row*1, *col*1) and lower right corner (*row*2, *col*2).

The above rectangle (with the red border) is defined by (row1, col1) = **(2, 1)** and (row2, col2) = **(4, 3)**, which contains sum = **8**.

Example:

```
Given matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]

sumRegion(2, 1, 4, 3) -> 8
sumRegion(1, 1, 2, 2) -> 11
sumRegion(1, 2, 2, 4) -> 12
```

Note:

1. You may assume that the matrix does not change.
2. There are many calls to *sumRegion* function.
3. You may assume that *row*1 ≤ *row*2 and *col*1 ≤ *col*2.

Solution 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

Solution 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

```
+-----+-+-------+     +--------+-----+     +-----+---------+     +-----+--------
+
|     | |       |     |        |     |     |     |         |     |     |
|
|     | |       |     |        |     |     |     |         |     |     |
|
+-----+-+       |     +--------+     |     |     |         |     +-----+
|
|     | |       |  =  |              |  +  |     |         |  -  |
|
+-----+-+       |     |              |     +-----+         |     |
|
|               |     |              |     |               |     |
|
|               |     |              |     |               |     |
|
+---------------+     +--------------+     +---------------+     +--------------
+

   sums[i][j]      =    sums[i-1][j]    +     sums[i][j-1]    -   sums[i-1][j-1]
+

                        matrix[i-1][j-1]
```

So, we use the same idea to find the specific area's sum.

```
+---------------+   +---------------+   +---------------+   +---------------+   +-
-------------+
|               |   |         |    |   |   |           |   |         |    |   |
|          |
|   (r1,c1)     |   |         |    |   |   |           |   |         |    |   |
|          |
|   +------+    |   |         |    |   |   |           |   +---------+    |   +-
--+          |
|   |      |    | = |         |    | - |   |           | - |      (r1,c2) | + |
(r1,c1)    |
|   |      |    |   |         |    |   |   |           |   |              |   |
|
|   +------+    |   +---------+    |   +---+           |   |              |   |
|
|        (r2,c2)|   |       (r2,c2)|   |   (r2,c1)     |   |              |   |
|
+---------------+   +---------------+   +---------------+   +---------------+   +-
-------------+
```

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

Solution 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

From Leetcoder.

Number of Islands II

A 2d grid map of `m` rows and `n` columns is initially filled with water. We may perform an *addLand* operation which turns the water at position (row, col) into a land. Given a list of positions to operate, **count the number of islands after each *addLand* operation**. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example:

Given `m = 3, n = 3`, `positions = [[0,0], [0,1], [1,2], [2,1]]`. Initially, the 2d grid `grid` is filled with water. (Assume 0 represents water and 1 represents land).

```
0 0 0
0 0 0
0 0 0
```

Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.

```
1 0 0
0 0 0   Number of islands = 1
0 0 0
```

Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.

```
1 1 0
0 0 0   Number of islands = 1
0 0 0
```

Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.

```
1 1 0
0 0 1   Number of islands = 2
0 0 0
```

Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land.

```
1 1 0
0 0 1   Number of islands = 3
0 1 0
```

We return the result as an array: `[1, 1, 2, 3]`

Challenge:

Can you do it in time complexity O(k log mn), where k is the length of the `positions`?

Solution 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);
```

```java
            }
            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;
    }
}
```

```
//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

Solution 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 belonging 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)`.

Remember that one island could have different `roots[node]` value for each node. Because `roots[node]` is the parent of the node, not the highest root of the island. To find the actually root, we have to climb up the tree by calling **findIsland** function.

Here I've attached my solution. There can be at least two improvements: `union by rank` & `pass compression`. However I suggest first finish the basis, then discuss the improvements.

Cheers!

```java
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<>();
    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];     // assume new point is isolated island
        roots[root] = root;             // add new island
        count++;

        for(int[] dir : dirs) {
            int x = p[0] + dir[0];
            int y = p[1] + dir[1];
            int nb = n * x + y;
            if(x < 0 || x >= m || y < 0 || y >= n || roots[nb] == -1) continue;

            int rootNb = findIsland(roots, nb);
            if(root != rootNb) {        // if neighbor is in another island
                roots[root] = rootNb;   // union two islands
                root = rootNb;          // current tree root = joined tree root
                count--;
            }
        }

        result.add(count);
    }
    return result;
}

public int findIsland(int[] roots, int id) {
    while(id != roots[id]) id = roots[id];
    return id;
}
```

Path Compression (Bonus)

If you have time, add one line to shorten the tree. The new runtime becomes: `19ms (95.94%)`.

```java
public int findIsland(int[] roots, int id) {
    while(id != roots[id]) {
        roots[id] = roots[roots[id]];   // only one line added
        id = roots[id];
    }
    return id;
}
```

written by yavinci original link here

Solution 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.

```java
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

Additive number is a string whose digits can form additive sequence.

A valid additive sequence should contain **at least** three numbers. Except for the first two numbers, each subsequent number in the sequence must be the sum of the preceding two.

For example:
`"112358"` is an additive number because the digits can form an additive sequence: `1, 1, 2, 3, 5, 8`.

```
1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8
```

`"199100199"` is also an additive number, the additive sequence is: `1, 99, 100, 199`.

```
1 + 99 = 100, 99 + 100 = 199
```

Note: Numbers in the additive sequence **cannot** have leading zeros, so sequence `1, 2, 03` or `1, 02, 3` is invalid.

Given a string containing only digits `'0'-'9'`, write a function to determine if it's an additive number.

Follow up:
How would you handle overflow for very large input integers?

Credits:
Special thanks to @jeantimex for adding this problem and creating all test cases.

Solution 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, sum, 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

Solution 2

The idea is quite straightforward:

1. Choose the first number `A`, it can be the leftmost `1` up to `i` digits. `i<=(L-1)/2` because the third number should be at least as long as the first number
2. Choose the second number `B`, it can be the leftmost `1` up to `j` digits excluding the first number. the limit for `j` is a little bit tricky, because we don't know whether `A` or `B` is longer. The remaining string (with length `L-j`) after excluding `A` and `B` should have a length of at least max(length `A`, length `B`), where length `A` = `i` and length `B` = `j-i`, thus `L-j >= max(j-i, i)`
3. Calls the recursive checker function and returns true if passes the checker function, or continue to the next choice of `B` (`A`) until there is no more choice for `B` or `A`, in which case returns a false.

Here is the code in Java:

```java
    public boolean isAdditiveNumber(String num) {
        int L = num.length();

        // choose the first number A
        for(int i=1; i<=(L-1)/2; i++) {
            // A cannot start with a 0 if its length is more than 1
            if(num.charAt(0) == '0' && i>=2) break; //previous code: continue;

            // choose the second number B
            for(int j=i+1; L-j>=j-i && L-j>=i; j++) {
                // B cannot start with a 0 if its length is more than 1
                if(num.charAt(i) == '0' && j-i>=2) break; // previous: continue;

                long num1 = Long.parseLong(num.substring(0, i)); // A
                long num2 = Long.parseLong(num.substring(i, j)); // B
                String substr = num.substring(j); // remaining string

                if(isAdditive(substr, num1, num2)) return true; // return true if
passes isAdditive test
                // else continue; // continue for loop if does not pass isAdditiv
e test
            }
        }
        return false; // does not pass isAdditive test, thus is not additive
    }

    // Recursively checks if a string is additive
    public boolean isAdditive(String str, long num1, long num2) {
        if(str.equals("")) return true; // reaches the end of string means a yes

        long sum = num1+num2;
        String s = ((Long)sum).toString();
        if(!str.startsWith(s)) return false; // if string does not start with sum
of num1 and num2, returns false

        return isAdditive(str.substring(s.length()), num2, sum); // recursively c
hecks the remaining string
    }
```

written by GWTW original link here

Solution 3

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) {
            if (num.charAt(0) == '0' && i > 1) return false;
            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(0) == '0' && i > 1) return false;
        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(0) == '0' && i > 1) return false;
        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

From Leetcoder.

Range Sum Query - Mutable

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* ($i \le j$), inclusive.

The *update(i, val)* function modifies *nums* by updating the element at index *i* to *val*.

Example:

```
Given nums = [1, 3, 5]

sumRange(0, 2) -> 9
update(1, 2)
sumRange(0, 2) -> 8
```

Note:

1. The array is only modifiable by the *update* function.
2. You may assume the number of calls to *update* and *sumRange* function is distributed evenly.

Solution 1

```java
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

Solution 2

The idea is using “buckets”. 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 “sum” value in that bucket accordingly. The operation takes O(1) time.

Summation is a little complicated. In the above example, let’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 don’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 “+” operations. If you change the size of buckets, the number of saved “+” operations will be different. The questions is:

What is the best size that can save the most “+” 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** “+” 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

Solution 3

This is to share the explanation of the BIT and the meaning of the bit operations.

```
public class NumArray {
    /**
     * Binary Indexed Trees (BIT or Fenwick tree):
     * https://www.topcoder.com/community/data-science/data-science-
     * tutorials/binary-indexed-trees/
     *
     * Example: given an array a[0]...a[7], we use a array BIT[9] to
     * represent a tree, where index [2] is the parent of [1] and [3], [6]
     * is the parent of [5] and [7], [4] is the parent of [2] and [6], and
     * [8] is the parent of [4]. I.e.,
     *
     * BIT[] as a binary tree:
     *            _______________*
     *            ______*
     *            __*     __*
     *            *   *   *   *
     * indices: 0 1 2 3 4 5 6 7 8
     *
     * BIT[i] = ([i] is a left child) ? the partial sum from its left most
     * descendant to itself : the partial sum from its parent (exclusive) to
     * itself. (check the range of "__").
     *
     * Eg. BIT[1]=a[0], BIT[2]=a[1]+BIT[1]=a[1]+a[0], BIT[3]=a[2],
     * BIT[4]=a[3]+BIT[3]+BIT[2]=a[3]+a[2]+a[1]+a[0],
     * BIT[6]=a[5]+BIT[5]=a[5]+a[4],
     * BIT[8]=a[7]+BIT[7]+BIT[6]+BIT[4]=a[7]+a[6]+...+a[0], ...
     *
     * Thus, to update a[1]=BIT[2], we shall update BIT[2], BIT[4], BIT[8],
     * i.e., for current [i], the next update [j] is j=i+(i&-i) //double the
     * last 1-bit from [i].
     *
     * Similarly, to get the partial sum up to a[6]=BIT[7], we shall get the
     * sum of BIT[7], BIT[6], BIT[4], i.e., for current [i], the next
     * summand [j] is j=i-(i&-i) // delete the last 1-bit from [i].
     *
     * To obtain the original value of a[7] (corresponding to index [8] of
     * BIT), we have to subtract BIT[7], BIT[6], BIT[4] from BIT[8], i.e.,
     * starting from [idx-1], for current [i], the next subtrahend [j] is
     * j=i-(i&-i), up to j==idx-(idx&-idx) exclusive. (However, a quicker
     * way but using extra space is to store the original array.)
     */

    int[] nums;
    int[] BIT;
    int n;

    public NumArray(int[] nums) {
        this.nums = nums;

        n = nums.length;
        BIT = new int[n + 1];
        for (int i = 0; i < n; i++)
            init(i, nums[i]);
```

```
            init(i, nums[i]);
    }

    public void init(int i, int val) {
        i++;
        while (i <= n) {
            BIT[i] += val;
            i += (i & -i);
        }
    }

    void update(int i, int val) {
        int diff = val - nums[i];
        nums[i] = val;
        init(i, diff);
    }

    public int getSum(int i) {
        int sum = 0;
        i++;
        while (i > 0) {
            sum += BIT[i];
            i -= (i & -i);
        }
        return sum;
    }

    public int sumRange(int i, int j) {
        return getSum(j) - getSum(i - 1);
    }
}

// Your NumArray object will be instantiated and called as such:
// NumArray numArray = new NumArray(nums);
// numArray.sumRange(0, 1);
// numArray.update(1, 10);
// numArray.sumRange(1, 2);
```

written by rikimberley original link here

From Leetcoder.

Range Sum Query 2D - Mutable

Given a 2D matrix *matrix*, find the sum of the elements inside the rectangle defined by its upper left corner (*row*1, *col*1) and lower right corner (*row*2, *col*2).

The above rectangle (with the red border) is defined by (row1, col1) = **(2, 1)** and (row2, col2) = **(4, 3)**, which contains sum = **8**.

Example:

```
Given matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]

sumRegion(2, 1, 4, 3) -> 8
update(3, 2, 2)
sumRegion(2, 1, 4, 3) -> 10
```

Note:

1. The matrix is only modifiable by the *update* function.
2. You may assume the number of calls to *update* and *sumRegion* function is distributed evenly.
3. You may assume that $row1 \le row2$ and $col1 \le col2$.

Solution 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

Solution 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

Solution 3

I have written both the Quad tree based solution and the indexed tree based solution for c++.

Both are very straight-forward. I have made some mistake for my previous analysis of the quad-tree solution. The indexed tree solution is more efficient in general.

Method 1: Quad-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(max(m, n)) 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 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

From Leetcoder.

Best Time to Buy and Sell Stock with Cooldown

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

- You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
- After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

```
prices = [1, 2, 3, 0, 2]
maxProfit = 3
transactions = [buy, sell, cooldown, buy, sell]
```

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 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

Solution 2

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

Solution 3

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

From Leetcoder.

Minimum Height Trees

For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Format
The graph contains `n` nodes which are labeled from `0` to `n - 1`. You will be given the number `n` and a list of undirected `edges` (each edge is a pair of labels).

You can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]` and thus will not appear together in `edges`.

Example 1:

Given `n = 4`, `edges = [[1, 0], [1, 2], [1, 3]]`

```
        0
        |
        1
       / \
      2   3
```

return `[1]`

Example 2:

Given `n = 6`, `edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]`

```
     0  1  2
      \ | /
        3
        |
        4
        |
        5
```

return `[3, 4]`

1. How many MHTs can a graph have at most?

Note:

(1) According to the definition of tree on Wikipedia: "a tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree."

(2) The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

See here for a better view

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

```
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

```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

Solution 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> 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

Solution 3

I am sharing two of my solutions, one is based on the longest path, and the other is related to Tree DP.

Longest Path

It is easy to see that the root of an MHT has to be the middle point (or two middle points) of the longest path of the tree. Though multiple longest paths can appear in an unrooted tree, they must share the same middle point(s).

Computing the longest path of a unrooted tree can be done, in O(n) time, by tree dp, or simply 2 tree traversals (dfs or bfs). The following is some thought of the latter.

Randomly select a node x as the root, do a dfs/bfs to find the node y that has the longest distance from x. Then y must be one of the endpoints on some longest path. Let y the new root, and do another dfs/bfs. Find the node z that has the longest distance from y.

Now, the path from y to z is the longest one, and thus its middle point(s) is the answer. Java Solution

Tree DP

Alternatively, one can solve this problem directly by tree dp. Let dp[i] be the height of the tree when the tree root is i. We compute dp[0] ... dp[n - 1] by tree dp in a dfs manner.

Arbitrarily pick a node, say node 0, as the root, and do a dfs. When we reach a node u, and let T be the subtree by removing all u's descendant (see the right figure below). We maintain a variable acc that keeps track of the length of the longest path in T with one endpoint being u. Then dp[u] = max(height[u], acc) Note, acc is 0 for the root of the tree.

```
          |                 |
          .                 .
         /|\               /|\
        * u *             * u *
         /|\
        / | \
       *  v  *
```

. denotes a single node, and * denotes a subtree (possibly empty).

Now it remains to calculate the new acc for any of u's child, v. It is easy to see that the new acc is the max of the following

1. acc + 1 --- extend the previous path by edge uv;
2. max(height[v'] + 2), where v != v' --- see below for an example.

```
        u
       /|
      / |
    v'  v
    |
    .
    .
    .
    |
    .
```

In fact, the second case can be computed in O(1) time instead of spending a time proportional to the degree of u. Otherwise, the runtime can be quadratic when the degree of some node is Omega(n). The trick here is to maintain two heights of each node, the largest height (the conventional height), and the second largest height (the height of the node after removing the branch w.r.t. the largest height).

Therefore, after the dfs, all dp[i]'s are computed, and the problem can be answered trivially. The total runtime is still O(n). Java Solution

written by lixx2100 original link here

From Leetcoder.

Sparse Matrix Multiplication

Given two sparse matrices **A** and **B**, return the result of **AB**.

You may assume that **A**'s column number is equal to **B**'s row number.

Example:

```
A = [
  [ 1, 0, 0],
  [-1, 0, 3]
]

B = [
  [ 7, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 1 ]
]


     |  1 0 0 |   | 7 0 0 |   |  7 0 0 |
AB = | -1 0 3 | x | 0 0 0 | = | -7 0 3 |
                  | 0 0 1 |
```

Solution 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

Solution 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):

```java
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):

```python
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):

```java
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):

```python
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):

```java
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

Solution 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

From Leetcoder.

Burst Balloons

Given `n` balloons, indexed from `0` to `n-1`. Each balloon is painted with a number on it represented by array `nums`. You are asked to burst all the balloons. If the you burst balloon `i` you will get `nums[left] * nums[i] * nums[right]` coins. Here `left` and `right` are adjacent indices of `i`. After the burst, the `left` and `right` then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note:
(1) You may imagine `nums[-1] = nums[n] = 1`. They are not real therefore you can not burst them.
(2) 0 ≤ `n` ≤ 500, 0 ≤ `nums[i]` ≤ 100

Example:

Given `[3, 1, 5, 8]`

Return `167`

```
 nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
coins =  3*1*5      +  3*5*8    +  1*3*8      + 1*8*1   = 167
```

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

See here for a better view

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 then 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 in dp problem analysis. That is reverse thinking. Like I said the coins you get for a balloon 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 affect 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

Solution 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

Solution 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

From Leetcoder.

Super Ugly Number

Write a program to find the nth super ugly number.

Super ugly numbers are positive numbers whose all prime factors are in the given prime list `primes` of size `k`. For example, `[1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32]` is the sequence of the first 12 super ugly numbers given `primes` = `[2, 7, 13, 19]` of size 4.

Note:
(1) `1` is a super ugly number for any given `primes`.
(2) The given numbers in `primes` are in ascending order.
(3) 0 k ≤ 100, 0 n ≤ 10^6, 0 primes[i]

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 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

Solution 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

Solution 3

Solution 1 ... ~1570 ms

Using generators and `heapq.merge`. Too bad there's no `itertools.unique`.

```
def nthSuperUglyNumber(self, n, primes):
    uglies = [1]
    def gen(prime):
        for ugly in uglies:
            yield ugly * prime
    merged = heapq.merge(*map(gen, primes))
    while len(uglies) < n:
        ugly = next(merged)
        if ugly != uglies[-1]:
            uglies.append(ugly)
    return uglies[-1]
```

Solution 2 ... ~1400 ms

Same thing done differently and it's a bit faster.

```
def nthSuperUglyNumber(self, n, primes):
    uglies = [1]
    merged = heapq.merge(*map(lambda p: (u*p for u in uglies), primes))
    uniqed = (u for u, _ in itertools.groupby(merged))
    map(uglies.append, itertools.islice(uniqed, n-1))
    return uglies[-1]
```

written by StefanPochmann original link here

From Leetcoder.

Binary Tree Vertical Order Traversal

Given a binary tree, return the *vertical order* traversal of its nodes' values. (ie, from top to bottom, column by column).

If two nodes are in the same row and column, the order should be from**left to right**.

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

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

return its vertical order traversal as:

```
[
  [9],
  [3,15],
  [20],
  [7]
]
```

Given binary tree `[3,9,20,4,5,2,7]`,

```
    _3_
   /   \
  9    20
 / \   / \
4   5 2   7
```

return its vertical order traversal as:

```
[
  [4],
  [9],
  [3,5,2],
  [20],
  [7]
]
```

Solution 1

The following solution takes `5ms`.

- BFS, put `node`, `col` into queue at the same time
- Every left child access `col - 1` while right child `col + 1`
- This maps `node` into different `col` buckets
- Get `col` boundary `min` and `max` on the fly
- Retrieve `result` from `cols`

Note that `TreeMap` version takes `9ms`.

Here is an example of `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]`. Notice that every child access changes one column bucket id. So `12` actually goes ahead of `11`.

```java
public List<List<Integer>> verticalOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    if(root == null) return res;

    Map<Integer, ArrayList<Integer>> map = new HashMap<>();
    Queue<TreeNode> q = new LinkedList<>();
    Queue<Integer> cols = new LinkedList<>();

    q.add(root);
    cols.add(0);

    int min = 0, max = 0;
    while(!q.isEmpty()) {
        TreeNode node = q.poll();
        int col = cols.poll();
        if(!map.containsKey(col)) map.put(col, new ArrayList<Integer>());
        map.get(col).add(node.val);

        if(node.left != null) {
            q.add(node.left);
            cols.add(col - 1);
            if(col <= min) min = col - 1;
        }
        if(node.right != null) {
            q.add(node.right);
            cols.add(col + 1);
            if(col >= max) max = col + 1;
        }
    }

    for(int i = min; i <= max; i++) {
        res.add(map.get(i));
    }

    return res;
}
```

written by yavinci original link here

Solution 2

```
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

Solution 3

```
private int min = 0, max = 0;
public List<List<Integer>> verticalOrder(TreeNode root) {
    List<List<Integer>> list = new ArrayList<>();
    if(root == null)    return list;
    computeRange(root, 0);
    for(int i = min; i <= max; i++) list.add(new ArrayList<>());
    Queue<TreeNode> q = new LinkedList<>();
    Queue<Integer> idx = new LinkedList<>();
    idx.add(-min);
    q.add(root);
    while(!q.isEmpty()){
        TreeNode node = q.poll();
        int i = idx.poll();
        list.get(i).add(node.val);
        if(node.left != null){
            q.add(node.left);
            idx.add(i - 1);
        }
        if(node.right != null){
            q.add(node.right);
            idx.add(i + 1);
        }
    }
    return list;
}
private void computeRange(TreeNode root, int idx){
    if(root == null)    return;
    min = Math.min(min, idx);
    max = Math.max(max, idx);
    computeRange(root.left, idx - 1);
    computeRange(root.right, idx + 1);
}
```

There is no difference when using HashMap. Since by using HashMap it need keep track of min and max as well, I'd rather directly insert into list by computing min and max in advance.

written by Jinx_boom original link here

From Leetcoder.

Count of Smaller Numbers After Self

You are given an integer array *nums* and you have to return a new *counts* array. The *counts* array has the property where `counts[i]` is the number of smaller elements to the right of `nums[i]`.

Example:

```
Given nums = [5, 2, 6, 1]

To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.
```

Return the array `[2, 1, 1, 0]`.

Solution 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

Solution 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

Solution 3

Every node will maintain a val **sum** recording the total of number on it's left bottom side, **dup** counts the duplication. For example, [3, 2, 2, 6, 1], from back to beginning,we would have:

```
1(0, 1)
     \
     6(3, 1)
     /
   2(0, 2)
       \
        3(0, 1)
```

When we try to insert a number, the total number of smaller number would be **adding dup and sum of the nodes where we turn right** . for example, if we insert 5, it should be inserted on the way down to the right of 3, the nodes where we turn right is 1(0,1), 2,(0,2), 3(0,1), so the answer should be (0 + 1)+(0 + 2)+ (0 + 1) = 4

if we insert 7, the right-turning nodes are 1(0,1), 6(3,1), so answer should be (0 + 1) + (3 + 1) = 5

```java
public class Solution {
    class Node {
        Node left, right;
        int val, sum, dup = 1;
        public Node(int v, int s) {
            val = v;
            sum = s;
        }
    }
    public List<Integer> countSmaller(int[] nums) {
        Integer[] ans = new Integer[nums.length];
        Node root = null;
        for (int i = nums.length - 1; i >= 0; i--) {
            root = insert(nums[i], root, ans, i, 0);
        }
        return Arrays.asList(ans);
    }
    private Node insert(int num, Node node, Integer[] ans, int i, int preSum) {
        if (node == null) {
            node = new Node(num, 0);
            ans[i] = preSum;
        } else if (node.val == num) {
            node.dup++;
            ans[i] = preSum + node.sum;
        } else if (node.val > num) {
            node.sum++;
            node.left = insert(num, node.left, ans, i, preSum);
        } else {
            node.right = insert(num, node.right, ans, i, preSum + node.dup + node
.sum);
        }
        return node;
    }
}
```

written by mayanist original link here

From Leetcoder.

Remove Duplicate Letters

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example:

Given `"bcabc"`
Return `"abc"`

Given `"cbacdcbc"`
Return `"acdb"`

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 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++) {
            if (s.charAt(i) < s.charAt(pos)) pos = i;
            if (--cnt[s.charAt(i) - '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

Solution 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

---

```java
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

Solution 3

for "cbacdcbc", we counts each letter's index:

```
a----2
b----1,6
c----0,3,5,7
d----4
```

we go from a to d, to find the first letter who has a index smaller than the largest index of the rest. Here, index 2 of letter a is smaller than 6, 7, 4, so we first pick a; then we remove all index smaller than 2, and we have:

```
b----6
c----3,5,7
d----4
```

the next round we pick c not b, why ? cuz 6 of b is larger than 4, but 3 of c is smaller than 4 and 6.

```
b---6
d---4
```

then we pick d and b to form "acdb"

O(n) time to count index, and as we only have 26 letters, it's about O(26 * 26) to find a candidate letter and O(n) time to remove all index. So I think the running time is O(n).

```java
public class Solution {
    public String removeDuplicateLetters(String s) {
        HashMap<Character, ArrayList<Integer>> counts = new HashMap<Character, Ar
rayList<Integer>>();
        ArrayList<Character> keys = new ArrayList<Character>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (!counts.containsKey(c)) {
                counts.put(c, new ArrayList<Integer>());
                keys.add(c);
            }
            counts.get(c).add(i);
        }
        Collections.sort(keys);
        StringBuilder sb = new StringBuilder();
        while (!counts.isEmpty()) {
            boolean found = true;
            for (int i = 0; i < keys.size(); i++) {
                int index = counts.get(keys.get(i)).get(0);
                for (int j = 0; j < keys.size(); j++) {
                    ArrayList<Integer> count = counts.get(keys.get(j));
                    if (count.get(count.size() - 1) < index) {
                        found = false;
                        break;
                    }
                }
                if (found) {
                    sb.append(keys.get(i));
                    counts.remove(keys.get(i));
                    keys.remove(i);
                    for (int j = 0; j < keys.size(); j++) {
                        ArrayList<Integer> count = counts.get(keys.get(j));
                        while (count.get(0) < index) {
                            count.remove(0);
                        }
                    }
                    break;
                }
                found = true;
            }
        }
        return sb.toString();
    }
}
```

written by mayanist original link here

From Leetcoder.

Shortest Distance from All Buildings

You want to build a house on an *empty* land which reaches all buildings in the shortest amount of distance. You can only move up, down, left and right. You are given a 2D grid of values **0**, **1** or **2**, where:

- Each **0** marks an empty land which you can pass by freely.
- Each **1** marks a building which you cannot pass through.
- Each **2** marks an obstacle which you cannot pass through.

For example, given three buildings at `(0,0)`, `(0,4)`, `(2,2)`, and an obstacle at `(0,2)`:

```
1 - 0 - 2 - 0 - 1
|   |   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |   |
0 - 0 - 1 - 0 - 0
```

The point `(1,2)` is an ideal empty land to build a house, as the total travel distance of 3+3+1=7 is minimal. So return 7.

Note:
There will be at least one building. If it is not possible to build such house according to the above rules, return -1.

Solution 1

I also tested the other three C++ solutions posted so far, they took 340-1812 ms. I think mine is faster because I don't use a fresh "`visited`" for each BFS. Instead, I walk only onto the cells that were reachable from all previous buildings. From the first building I only walk onto cells where `grid` is 0, and make them -1. From the second building I only walk onto cells where `grid` is -1, and I make them -2. And so on.

```
int shortestDistance(vector<vector<int>> grid) {
    int m = grid.size(), n = grid[0].size();
    auto total = grid;
    int walk = 0, mindist, delta[] = {0, 1, 0, -1, 0};
    for (int i=0; i<m; ++i) {
        for (int j=0; j<n; ++j) {
            if (grid[i][j] == 1) {
                mindist = -1;
                auto dist = grid;
                queue<pair<int, int>> q;
                q.emplace(i, j);
                while (q.size()) {
                    auto ij = q.front();
                    q.pop();
                    for (int d=0; d<4; ++d) {
                        int i = ij.first + delta[d];
                        int j = ij.second + delta[d+1];
                        if (i >= 0 && i < m && j >= 0 && j < n && grid[i][j] == w
alk) {
                            grid[i][j]--;
                            dist[i][j] = dist[ij.first][ij.second] + 1;
                            total[i][j] += dist[i][j] - 1;
                            q.emplace(i, j);
                            if (mindist < 0 || mindist > total[i][j])
                                mindist = total[i][j];
                        }
                    }
                }
                walk--;
            }
        }
    }
    return mindist;
}
```

written by StefanPochmann original link here

Solution 2

Inspired by previous solution. The main idea is the following:

Traverse the matrix. For each building, use BFS to compute the shortest distance from each '0' to this building. After we do this for all the buildings, we can get the sum of shortest distance from every '0' to all reachable buildings. This value is stored in 'distance[][]'. For example, if grid[2][2] == 0, distance[2][2] is the sum of shortest distance from this block to all reachable buildings. Time complexity: O(number of 1)*O(number of 0) ~ O(m^2n^2)*

We also count how many building each '0' can be reached. It is stored in reach[][]. This can be done during the BFS. We also need to count how many total buildings are there in the matrix, which is stored in 'buildingNum'.

Finally, we can traverse the distance[][] matrix to get the point having shortest distance to all buildings. O(m*n)

The total time complexity will be O(m^2*n^2), which is quite high!. Please let me know if I did the analysis wrong or you have better solution.

```
public class Solution {
    public int shortestDistance(int[][] grid) {
        if (grid == null || grid[0].length == 0) return 0;
        final int[] shift = new int[] {0, 1, 0, -1, 0};

        int row  = grid.length, col = grid[0].length;
        int[][] distance = new int[row][col];
        int[][] reach = new int[row][col];
        int buildingNum = 0;

        for (int i = 0; i < row; i++) {
            for (int j =0; j < col; j++) {
                if (grid[i][j] == 1) {
                    buildingNum++;
                    Queue<int[]> myQueue = new LinkedList<int[]>();
                    myQueue.offer(new int[] {i,j});

                    boolean[][] isVisited = new boolean[row][col];
                    int level = 1;

                    while (!myQueue.isEmpty()) {
                        int qSize = myQueue.size();
                        for (int q = 0; q < qSize; q++) {
                            int[] curr = myQueue.poll();

                            for (int k = 0; k < 4; k++) {
                                int nextRow = curr[0] + shift[k];
                                int nextCol = curr[1] + shift[k + 1];

                                if (nextRow >= 0 && nextRow < row && nextCol >= 0
&& nextCol < col
                                    && grid[nextRow][nextCol] == 0 && !isVisited[
nextRow][nextCol]) {
                                        //The shortest distance from [nextRow][ne
```

```
xtCol] to this building
                                        // is 'level'.
                                        distance[nextRow][nextCol] += level;
                                        reach[nextRow][nextCol]++;

                                        isVisited[nextRow][nextCol] = true;
                                        myQueue.offer(new int[] {nextRow, nextCol
});
                                    }
                                }
                            }
                            level++;
                        }
                    }
                }
            }

            int shortest = Integer.MAX_VALUE;
            for (int i = 0; i < row; i++) {
                for (int j = 0; j < col; j++) {
                    if (grid[i][j] == 0 && reach[i][j] == buildingNum) {
                        shortest = Math.min(shortest, distance[i][j]);
                    }
                }
            }

            return shortest == Integer.MAX_VALUE ? -1 : shortest;


        }
    }
```

written by shuoshankou original link here

Solution 3

```
    int shortestDistance(vector<vector<int>>& grid) {
    const int row = grid.size();
    if (0 == row) return -1;
    const int col = grid[0].size();

    vector<vector<int> > distance(row, vector<int>(col, 0));
    vector<vector<int>> reach(row, vector<int>(col, 0));
    int building = 0, res = INT_MAX;

    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++) {
        // check from the building node, extend to all 0 node with distance
        if (1 == grid[i][j]) {
            ++building;
            int dist = 0;
            vector<vector<bool>> visited(row, vector<bool>(col, false));
            queue<pair<int, int>> curLevel, nextLevel;
            curLevel.emplace(i, j);
            // bfs search for each current building
            while (!curLevel.empty()) {
                ++dist;
                while (!curLevel.empty()) {
                    pair<int, int> cur = curLevel.front();
                    curLevel.pop();
                    int x = cur.first, y = cur.second;
                    ++reach[x][y];
                    vector<pair<int, int>> dirs = {{1, 0}, {-1, 0}, {0, 1}, {0,
-1}};
                    for (auto dir : dirs) {
                        int i = x + dir.first, j = y + dir.second;
                        if (i >= 0 && i < grid.size() && j >= 0 && j < grid[0].si
ze() && 0 == grid[i][j] && !visited[i][j])
                        {
                            distance[i][j] += dist;
                            nextLevel.emplace(i, j);
                            visited[i][j] = true;
                        }
                    }
                }
                swap(curLevel, nextLevel);
            }
        }
    }
for (int i = 0; i < row; i++) {
    for (int j = 0; j < col; j++) {
        if (0 == grid[i][j] && reach[i][j] == building) {
            res = min(res, distance[i][j]);
        }
    }
}
return res == INT_MAX ? -1 : res;
```

}

written by lchen77 original link here

From Leetcoder.

Maximum Product of Word Lengths

Given a string array `words`, find the maximum value of `length(word[i]) * length(word[j])` where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

Example 1:

Given `["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]`
Return `16`
The two words can be `"abcw", "xtfn"`.

Example 2:

Given `["a", "ab", "abc", "d", "cd", "bcd", "abcd"]`
Return `4`
The two words can be `"ab", "cd"`.

Example 3:

Given `["a", "aa", "aaa", "aaaa"]`
Return `0`
No such pair of words.

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

```java
public class Solution {
    public int maxProduct(String[] words) {
        int max = 0;

        Arrays.sort(words, new Comparator<String>(){
            public int compare(String a, String b){
                return b.length() - a.length();
            }
        });

        int[] masks = new int[words.length]; // alphabet masks

        for(int i = 0; i < masks.length; i++){
            for(char c: words[i].toCharArray()){
                masks[i] |= 1 << (c - 'a');
            }
        }

        for(int i = 0; i < masks.length; i++){
            if(words[i].length() * words[i].length() <= max) break; //prunning
            for(int j = i + 1; j < masks.length; j++){
                if((masks[i] & masks[j]) == 0){
                    max = Math.max(max, words[i].length() * words[j].length());
                    break; //prunning
                }
            }
        }

        return max;
    }
}
```

written by cc9208 original link here

Solution 2

Same algorithm as most, just written a bit shorter.

```
int maxProduct(vector<string>& words) {
    vector<int> mask(words.size());
    int result = 0;
    for (int i=0; i<words.size(); ++i) {
        for (char c : words[i])
            mask[i] |= 1 << (c - 'a');
        for (int j=0; j<i; ++j)
            if (!(mask[i] & mask[j]))
                result = max(result, int(words[i].size() * words[j].size()));
    }
    return result;
}
```

Update: Here's an O(n+N) variation, where N is the total number of characters in all words. Thanks to junhuangli for the suggestion.

```
int maxProduct(vector<string>& words) {
    unordered_map<int,int> maxlen;
    for (string word : words) {
        int mask = 0;
        for (char c : word)
            mask |= 1 << (c - 'a');
        maxlen[mask] = max(maxlen[mask], (int) word.size());
    }
    int result = 0;
    for (auto a : maxlen)
        for (auto b : maxlen)
            if (!(a.first & b.first))
                result = max(result, a.second * b.second);
    return result;
}
```

Or: (thanks to junhuangli's further comment)

```
int maxProduct(vector<string>& words) {
    unordered_map<int,int> maxlen;
    int result = 0;
    for (string word : words) {
        int mask = 0;
        for (char c : word)
            mask |= 1 << (c - 'a');
        maxlen[mask] = max(maxlen[mask], (int) word.size());
        for (auto maskAndLen : maxlen)
            if (!(mask & maskAndLen.first))
                result = max(result, (int) word.size() * maskAndLen.second);
    }
    return result;
}
```

written by StefanPochmann original link here

Solution 3

Sort the vector first according to the length of string. Then use some early pruning to fasten the process.

The worst cases would still be O(N^2). It's faster than most O(N^2) solutions. I don't know whether we can do better. (Binary Search Seems Not work here) Any comments is welcomed.

Update: We can use counting sort to improve the time complexity of sorting to O(N).

```
class Solution {
public:
    int maxProduct(vector<string>& words) {
        int s=words.size();
        if(s==0) return 0;
        vector<int> bit(s,0);
        sort(words.begin(), words.end(), compare);  //sort the vector from longes
t to shortest
        for(int i=0; i<s; i++){ //bit manipulation
            for(int j=0; j<words[i].size(); j++) bit[i] |=(1<<(words[i][j]-'a'));

        }
        int maxlength=0;
        for(int i=0; i<s-1; i++){
            int si=words[i].size();
            if(si*si<=maxlength) break; //early pruning
            for(int j=i+1; j<s; j++){
                int sj=words[j].size();
                if(si*sj<=maxlength) break;  //early pruning
                if((bit[i]&bit[j])==0) maxlength=si*sj;
            }
        }
        return maxlength;
    }
    static bool compare(string a, string b){
        return a.size()>b.size();
    }
};
```

written by zjh08177 original link here

From Leetcoder.

Bulb Switcher

There are n bulbs that are initially off. You first turn on all the bulbs. Then, you turn off every second bulb. On the third round, you toggle every third bulb (turning on if it's off or turning off if it's on). For the nth round, you only toggle the last bulb. Find how many bulbs are on after n rounds.

Example:

```
Given n = 3.

At first, the three bulbs are [off, off, off].
After first round, the three bulbs are [on, on, on].
After second round, the three bulbs are [on, off, on].
After third round, the three bulbs are [on, off, off].

So you should return 1, because there is only one bulb is on.
```

Solution 1

```
int bulbSwitch(int n) {
    return sqrt(n);
}
```

A bulb ends up on iff it is switched an odd number of times.

Call them bulb 1 to bulb n. Bulb i is switched in round d if and only if d divides i. So bulb i ends up on if and only if it has an odd number of divisors.

Divisors come in pairs, like i=12 has divisors 1 and 12, 2 and 6, and 3 and 4. Except when i is a square, like 36 has divisors 1 and 36, 2 and 18, 3 and 12, 4 and 9, and double divisor 6. So bulb i ends up on if and only if i is a square.

So just count the square numbers.

Let R = int(sqrt(n)). That's the root of the largest square in the range [1,n]. And 1 is the smallest root. So you have the roots from 1 to R, that's R roots. Which correspond to the R squares. So int(sqrt(n)) is the answer. (C++ does the conversion to int automatically, because of the specified return type).

written by StefanPochmann original link here

Solution 2

```
int bulbSwitch(int n) {
    int counts = 0;

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

    return counts;
}
```

Explanation:
A light will be toggled only during the round of its factors, e.g. number 6 light will be toggled at 1,2,3,6 and light 12 will be toggled at 1,2,3,4,6,12. The final state of a light is on and off only depends on if the number of its factor is odd or even. If odd, the light is on and if even the light is off. The number of one number's factor is odd if and only if it is a perfect square!
So we will only need to loop to find all the perfect squares that are smaller than n!

written by larasieh original link here

Solution 3

```
// For prime numbers, they must be off because we can reach them only twice (The f
irst round and their own round).
/* For other numbers, if we can reach them odd times, then they are on; otherwise,
they are off. So only
 those numbers who have square root will be reached odd times and there are sqrt(
n) of them because
 for every x > sqrt(n), x*x > n and thus should not be considered as the answer.
*/

 return (int)sqrt(n);
```

written by yular original link here

From Leetcoder.

Generalized Abbreviation

Write a function to generate the generalized abbreviations of a word.

Example:

Given word = `"word"`, return the following list (order does not matter):

```
["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1
", "1o2", "2r1", "3d", "w3", "4"]
```

Solution 1

The idea is: for every character, we can keep it or abbreviate it. To keep it, we add it to the current solution and carry on backtracking. To abbreviate it, we omit it in the current solution, but increment the count, which indicates how many characters have we abbreviated. When we reach the end or need to put a character in the current solution, and count is bigger than zero, we add the number into the solution.

```
  public List<String> generateAbbreviations(String word){
        List<String> ret = new ArrayList<String>();
        backtrack(ret, word, 0, "", 0);

        return ret;
    }

    private void backtrack(List<String> ret, String word, int pos, String cur, in
t count){
        if(pos==word.length()){
            if(count > 0) cur += count;
            ret.add(cur);
        }
        else{
            backtrack(ret, word, pos + 1, cur, count + 1);
            backtrack(ret, word, pos+1, cur + (count>0 ? count : "") + word.charA
t(pos), 0);
        }
    }
```

written by soymuybien original link here

Solution 2

```
public class Solution {
    public List<String> generateAbbreviations(String word) {
        List<String> res = new ArrayList<String>();
        int len = word.length();
        res.add(len==0 ? "" : String.valueOf(len));
        for(int i = 0 ; i < len ; i++)
            for(String right : generateAbbreviations(word.substring(i+1))){
                String leftNum = i > 0 ? String.valueOf(i) : "";
                res.add( leftNum + word.substring(i,i + 1) + right );
            }
        return res;
    }
}
```

written by caiqi8877 original link here

Solution 3

For each char `c[i]`, either abbreviate it or not.

1. Abbreviate: count accumulate `num` of abbreviating chars, but don't append it yet.
2. Not Abbreviate: append accumulated `num` as well as current char `c[i]`.
3. In the end append remaining `num`.
4. Using `StringBuilder` can decrease `36.4%` time.

This comes to the pattern I find powerful:

```
int len = sb.length(); // decision point
... backtracking logic ...
sb.setLength(len);     // reset to decision point
```

Similarly, check out remove parentheses and add operators.

```
public List<String> generateAbbreviations(String word) {
    List<String> res = new ArrayList<>();
    DFS(res, new StringBuilder(), word.toCharArray(), 0, 0);
    return res;
}

public void DFS(List<String> res, StringBuilder sb, char[] c, int i, int num) {
    int len = sb.length();
    if(i == c.length) {
        if(num != 0) sb.append(num);
        res.add(sb.toString());
    } else {
        DFS(res, sb, c, i + 1, num + 1);                // abbr c[i]

        if(num != 0) sb.append(num);                    // not abbr c[i]
        DFS(res, sb.append(c[i]), c, i + 1, 0);
    }
    sb.setLength(len);
}
```

written by yavinci original link here

Create Maximum Number

Given two arrays of length `m` and `n` with digits `0-9` representing two numbers. Create the maximum number of length `k from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the k digits. You should try to optimize your time and space complexity.`

Example 1:

nums1 = `[3, 4, 6, 5]`
nums2 = `[9, 1, 2, 5, 8, 3]`
k = `5`
return `[9, 8, 6, 5, 3]`

Example 2:

nums1 = `[6, 7]`
nums2 = `[6, 0, 4]`
k = `5`
return `[6, 7, 6, 0, 4]`

Example 3:

nums1 = `[3, 9]`
nums2 = `[8, 9]`
k = `3`
return `[9, 8, 9]`

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

Many of the posts have the same algorithm. In short we can first solve 2 simpler problem

1. Create the maximum number of one array
2. Create the maximum number of two array using all of their digits.

For an long and detailed explanation see my blog here.

The algorithm is O((m+n)^3) in the worst case. It runs in 22 ms.

Java

```
public int[] maxNumber(int[] nums1, int[] nums2, int k) {
    int n = nums1.length;
    int m = nums2.length;
    int[] ans = new int[k];
    for (int i = Math.max(0, k - m); i <= k && i <= n; ++i) {
        int[] candidate = merge(maxArray(nums1, i), maxArray(nums2, k - i), k);
        if (greater(candidate, 0, ans, 0)) ans = candidate;
    }
    return ans;
}
private int[] merge(int[] nums1, int[] nums2, int k) {
    int[] ans = new int[k];
    for (int i = 0, j = 0, r = 0; r < k; ++r)
        ans[r] = greater(nums1, i, nums2, j) ? nums1[i++] : nums2[j++];
    return ans;
}
public boolean greater(int[] nums1, int i, int[] nums2, int j) {
    while (i < nums1.length && j < nums2.length && nums1[i] == nums2[j]) {
        i++;
        j++;
    }
    return j == nums2.length || (i < nums1.length && nums1[i] > nums2[j]);
}
public int[] maxArray(int[] nums, int k) {
    int n = nums.length;
    int[] ans = new int[k];
    for (int i = 0, j = 0; i < n; ++i) {
        while (n - i + j > k && j > 0 && ans[j - 1] < nums[i]) j--;
        if (j < k) ans[j++] = nums[i];
    }
    return ans;
}
```

written by dietpepsi original link here

Solution 2

Python

```python
def maxNumber(self, nums1, nums2, k):

    def prep(nums, k):
        drop = len(nums) - k
        out = []
        for num in nums:
            while drop and out and out[-1] < num:
                out.pop()
                drop -= 1
            out.append(num)
        return out[:k]

    def merge(a, b):
        return [max(a, b).pop(0) for _ in a+b]

    return max(merge(prep(nums1, i), prep(nums2, k-i))
               for i in range(k+1)
               if i <= len(nums1) and k-i <= len(nums2))
```

Solved it on my own but now I see others already posted this idea. Oh well, at least it's short, particularly my `merge` function.

The last two lines can be combined, but I find it rather ugly and not worth it: `for i in range(max(k-len(nums2), 0), min(k, len(nums1))+1))`

Ruby

```ruby
def prep(nums, k)
  drop = nums.size - k
  out = [9]
  nums.each do |num|
    while drop > 0 && out[-1] < num
      out.pop
      drop -= 1
    end
    out << num
  end
  out[1..k]
end

def max_number(nums1, nums2, k)
  ([k-nums2.size, 0].max .. [nums1.size, k].min).map { |k1|
    parts = [prep(nums1, k1), prep(nums2, k-k1)]
    (1..k).map { parts.max.shift }
  }.max
end
```

C++

Translated it to C++ as well now. Not as short anymore, but still decent. And C++ allows different functions with the same name, so I chose to do that here to show how nicely the `maxNumber(nums1, nums2, k)` problem can be based on the problems `maxNumber(nums, k)` and `maxNumber(nums1, nums2)`, which would make fine problems on their own.

```
vector<int> maxNumber(vector<int>& nums1, vector<int>& nums2, int k) {
    int n1 = nums1.size(), n2 = nums2.size();
    vector<int> best;
    for (int k1=max(k-n2, 0); k1<=min(k, n1); ++k1)
        best = max(best, maxNumber(maxNumber(nums1, k1),
                                   maxNumber(nums2, k-k1)));
    return best;
}

vector<int> maxNumber(vector<int> nums, int k) {
    int drop = nums.size() - k;
    vector<int> out;
    for (int num : nums) {
        while (drop && out.size() && out.back() < num) {
            out.pop_back();
            drop--;
        }
        out.push_back(num);
    }
    out.resize(k);
    return out;
}

vector<int> maxNumber(vector<int> nums1, vector<int> nums2) {
    vector<int> out;
    while (nums1.size() + nums2.size()) {
        vector<int>& now = nums1 > nums2 ? nums1 : nums2;
        out.push_back(now[0]);
        now.erase(now.begin());
    }
    return out;
}
```

An alternative for `maxNumber(nums1, nums2)`:

```
vector<int> maxNumber(vector<int> nums1, vector<int> nums2) {
    vector<int> out;
    auto i1 = nums1.begin(), end1 = nums1.end();
    auto i2 = nums2.begin(), end2 = nums2.end();
    while (i1 != end1 || i2 != end2)
        out.push_back(lexicographical_compare(i1, end1, i2, end2) ? *i2++ : *i1++
);
    return out;
}
```

written by StefanPochmann original link here

Solution 3

To find the maximum ,we can enumerate how digits we should get from nums1 , we suppose it is i.

So , the digits from nums2 is K - i.

And we can use a stack to get the get maximum number(x digits) from one array.

OK, Once we choose two maximum subarray , we should combine it to the answer.

It is just like merger sort, but we should pay attention to the case: the two digital are equal.

we should find the digits behind it to judge which digital we should choose now.

In other words,we should judge which subarry is bigger than the other.

That's all.

If you have any question or suggest, I am happy you can comment on my blog : Create Maximum Number.

Thanks, merry christmas :)

update:use stack to find max sub array and it runs 21ms now.(thanks to @dietpepsi)

```
 /**  * Created by hrwhisper on 2015/11/23.  * http://www.hrwhisper.me/leetcode-c
reate-maximum-number/  */


public class Solution {
    public int[] maxNumber(int[] nums1, int[] nums2, int k) {
        int get_from_nums1 = Math.min(nums1.length, k);
        int[] ans = new int[k];
        for (int i = Math.max(k - nums2.length, 0); i <= get_from_nums1; i++) {
            int[] res1 = new int[i];
            int[] res2 = new int[k - i];
            int[] res = new int[k];
            res1 = solve(nums1, i);
            res2 = solve(nums2, k - i);
            int pos1 = 0, pos2 = 0, tpos = 0;

            while (res1.length > 0 && res2.length > 0 && pos1 < res1.length && po
s2 < res2.length) {
                if (compare(res1, pos1, res2, pos2))
                    res[tpos++] = res1[pos1++];
                else
                    res[tpos++] = res2[pos2++];
            }
            while (pos1 < res1.length)
                res[tpos++] = res1[pos1++];
            while (pos2 < res2.length)
                res[tpos++] = res2[pos2++];

            if (!compare(ans, 0, res, 0))
                ans = res;
```

```
        }

        return ans;
    }

    public boolean compare(int[] nums1, int start1, int[] nums2, int start2) {
        for (; start1 < nums1.length && start2 < nums2.length; start1++, start2++
) {
            if (nums1[start1] > nums2[start2]) return true;
            if (nums1[start1] < nums2[start2]) return false;
        }
        return start1 != nums1.length;
    }

    public int[] solve(int[] nums, int k) {
        int[] res = new int[k];
        int len = 0;
        for (int i = 0; i < nums.length; i++) {
            while (len > 0 && len + nums.length - i > k && res[len - 1] < nums[i]
) {
                len--;
            }
            if (len < k)
                res[len++] = nums[i];
        }
        return res;
    } }
```

written by hrwhisper original link here

From Leetcoder.

Coin Change

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

Example 1:
coins = `[1, 2, 5]`, amount = `11`
return `3` (11 = 5 + 5 + 1)

Example 2:
coins = `[2]`, amount = `3`
return `-1`.

Note:
You may assume that you have an infinite number of each kind of coin.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 1

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        int Max = amount + 1;
        vector<int> dp(amount + 1, Max);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int j = 0; j < coins.size(); j++) {
                if (coins[j] <= i) {
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1);
                }
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
};
```

written by wyattliu original link here

Solution 2

Recursive Method:

The idea is very classic dynamic programming: think of the last step we take. Suppose we have already found out the best way to sum up to amount `a`, then for the last step, we can choose any coin type which gives us a remainder `r` where `r = a-coins[i]` for all `i`'s. For every remainder, go through exactly the same process as before until either the remainder is 0 or less than 0 (meaning not a valid solution). With this idea, the only remaining detail is to store the minimum number of coins needed to sum up to `r` so that we don't need to recompute it over and over again.

Code in Java:

```
public class Solution {
public int coinChange(int[] coins, int amount) {
    if(amount<1) return 0;
    return helper(coins, amount, new int[amount]);
}

private int helper(int[] coins, int rem, int[] count) { // rem: remaining coins a
fter the last step; count[rem]: minimum number of coins to sum up to rem
    if(rem<0) return -1; // not valid
    if(rem==0) return 0; // completed
    if(count[rem-1] != 0) return count[rem-1]; // already computed, so reuse
    int min = Integer.MAX_VALUE;
    for(int coin : coins) {
        int res = helper(coins, rem-coin, count);
        if(res>=0 && res < min)
            min = 1+res;
    }
    count[rem-1] = (min==Integer.MAX_VALUE) ? -1 : min;
    return count[rem-1];
}
}
```

Iterative Method:

For the iterative solution, we think in bottom-up manner. Suppose we have already computed all the minimum counts up to `sum`, what would be the minimum count for `sum+1`?

Code in Java:

```java
public class Solution {
public int coinChange(int[] coins, int amount) {
    if(amount<1) return 0;
    int[] dp = new int[amount+1];
    int sum = 0;

    while(++sum<=amount) {
        int min = -1;
        for(int coin : coins) {
            if(sum >= coin && dp[sum-coin]!=-1) {
                int temp = dp[sum-coin]+1;
                min = min<0 ? temp : (temp < min ? temp : min);
            }
        }
        dp[sum] = min;
    }
    return dp[amount];
}
}
```

written by GWTW original link here

Solution 3

Java solution: O(amount) space, O(n*amount) time complexity

```
public class Solution {
    public int coinChange(int[] coins, int amount) {
        if (coins == null || coins.length == 0)
            return -1;

        if (amount <= 0)
            return 0;

        int dp[] = new int[amount + 1];
        for (int i = 1; i < dp.length; i++) {
            dp[i] = Integer.MAX_VALUE;
        }

        for (int am = 1; am < dp.length; am++) {
            for (int i = 0; i < coins.length; i++) {
                if (coins[i] <= am) {
                    int sub = dp[am - coins[i]];
                    if (sub != Integer.MAX_VALUE)
                        dp[am] = Math.min(sub + 1, dp[am]);
                }
            }
        }
        return dp[dp.length - 1] == Integer.MAX_VALUE ? -1 : dp[dp.length - 1];
    }
}
```

written by RomanShakhmanaev original link here

From Leetcoder.

Number of Connected Components in an Undirected Graph

Given `n` nodes labeled from `0` to `n - 1` and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 1:

```
0          3
|          |
1 --- 2    4
```

Given `n = 5` and `edges = [[0, 1], [1, 2], [3, 4]]`, return `2`.

Example 2:

```
0           4
|           |
1 --- 2 --- 3
```

Given `n = 5` and `edges = [[0, 1], [1, 2], [2, 3], [3, 4]]`, return `1`.

Note:
You can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]` and thus will not appear together in `edges`.

Solution 1

```
    private int[] father;
public int countComponents(int n, int[][] edges) {

    Set<Integer> set = new HashSet<Integer>();
    father = new int[n];
    for (int i = 0; i < n; i++) {
        father[i] = i;
    }
    for (int i = 0; i < edges.length; i++) {
         union(edges[i][0], edges[i][1]);
    }

    for (int i = 0; i < n; i++){
        set.add(find(i));
    }
    return set.size();
}

int find(int node) {
    if (father[node] == node) {
        return node;
    }
    father[node] = find(father[node]);
    return father[node];
}

void union(int node1, int node2) {
    father[find(node1)] = find(node2);
}
```

written by jmnjmnjmn original link here

Solution 2

DFS:

```
def countComponents(n, edges):
        def dfs(n, g, visited):
            if visited[n]:
                return
            visited[n] = 1
            for x in g[n]:
                dfs(x, g, visited)

        visited = [0] * n
        g = {x:[] for x in xrange(n)}
        for x, y in edges:
            g[x].append(y)
            g[y].append(x)

        ret = 0
        for i in xrange(n):
            if not visited[i]:
                dfs(i, g, visited)
                ret += 1

        return ret
```

BFS:

```
def countComponents(n, edges):
        g = {x:[] for x in xrange(n)}
        for x, y in edges:
            g[x].append(y)
            g[y].append(x)

        ret = 0
        for i in xrange(n):
            queue = [i]
            ret += 1 if i in g else 0
            for j in queue:
                if j in g:
                    queue += g[j]
                    del g[j]

        return ret
```

Union Find:

```
def countComponents(n, edges):
        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x])
            return parent[x]

        def union(xy):
            x, y = map(find, xy)
            if rank[x] < rank[y]:
                parent[x] = y
            else:
                parent[y] = x
                if rank[x] == rank[y]:
                    rank[x] += 1

        parent, rank = range(n), [0] * n
        map(union, edges)
        return len({find(x) for x in parent})
```

written by chris.zhang.336 original link here

Solution 3

Use the similar method as Number of Islands II. Use a findRoot function. See more details in https://leetcode.com/discuss/69572/easiest-java-solution-with-explanations

```
public int countComponents(int n, int[][] edges) {
    int res = n;

    int[] root = new int[n];
    for (int i = 0; i < n; i++) {
        root[i] = i;
    }
    for (int[] pair : edges) {
        int rootX = findRoot(root, pair[0]);
        int rootY = findRoot(root, pair[1]);
        if (rootX != rootY) {
            root[rootY] = rootX;
            res--;
        }
    }
    return res;
}
public int findRoot(int[] root, int i) {
    while (root[i] != i) i = root[i];
    return i;
}
```

written by beckychiu1988 original link here

From Leetcoder.

Wiggle Sort II

Given an unsorted array `nums`, reorder it such that `nums[0] nums[2] .`

Example:
(1) Given `nums = [1, 5, 1, 1, 6, 4]`, one possible answer is `[1, 4, 1, 5, 1, 6]`.
(2) Given `nums = [1, 3, 2, 2, 3, 1]`, one possible answer is `[2, 3, 1, 3, 1, 2]`.

Note:
You may assume all input has valid answer.

Follow Up:
Can you do it in O(n) time and/or in-place with O(1) extra space?

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

This post is mainly about what I call "virtual indexing" technique (I'm sure I'm not the first who came up with this, but I couldn't find anything about it, so I made up a name as well. If you know better, let me know).

Solution

```
void wiggleSort(vector<int>& nums) {
    int n = nums.size();

    // Find a median.
    auto midptr = nums.begin() + n / 2;
    nth_element(nums.begin(), midptr, nums.end());
    int mid = *midptr;

    // Index-rewiring.
    #define A(i) nums[(1+2*(i)) % (n|1)]

    // 3-way-partition-to-wiggly in O(n) time with O(1) space.
    int i = 0, j = 0, k = n - 1;
    while (j <= k) {
        if (A(j) > mid)
            swap(A(i++), A(j++));
        else if (A(j) < mid)
            swap(A(j), A(k--));
        else
            j++;
    }
}
```

Explanation

First I find a median using `nth_element`. That only guarantees O(n) **average** time complexity and I don't know about space complexity. I might write this myself using O(n) time and O(1) space, but that's not what I want to show here.

This post is about what comes **after** that. We can use three-way partitioning to arrange the numbers so that those *larger than* the median come first, then those *equal to* the median come next, and then those*smaller than* the median come last.

Ordinarily, you'd then use one more phase to bring the numbers to their final positions to reach the overall wiggle-property. But I don't know a nice O(1) space way for this. Instead, I embed this right into the partitioning algorithm. That algorithm simply works with indexes 0 to n-1 as usual, but sneaky as I am, I rewire those indexes where I want the numbers to actually end up. The partitioning-algorithm doesn't even know that I'm doing that, it just works like normal (it just uses `A(x)` instead of `nums[x]`).

Let's say `nums` is `[10,11,...,19]`. Then after nth_element and ordinary

partitioning, we might have this (15 is my median):

```
index:     0  1  2  3   4   5  6  7  8  9
number:   18 17 19 16  15  11 14 10 13 12
```

I rewire it so that the first spot has index 5, the second spot has index 0, etc, so that I might get this instead:

```
index:     5  0  6  1  7  2  8  3  9  4
number:   11 18 14 17 10 19 13 16 12 15
```

And 11 18 14 17 10 19 13 16 12 15 is perfectly wiggly. And the whole partitioning-to-wiggly-arrangement (everything after finding the median) only takes O(n) time and O(1) space.

If the above description is unclear, maybe this explicit listing helps:

Accessing `A(0)` actually accesses `nums[1]`.
Accessing `A(1)` actually accesses `nums[3]`.
Accessing `A(2)` actually accesses `nums[5]`.
Accessing `A(3)` actually accesses `nums[7]`.
Accessing `A(4)` actually accesses `nums[9]`.
Accessing `A(5)` actually accesses `nums[0]`.
Accessing `A(6)` actually accesses `nums[2]`.
Accessing `A(7)` actually accesses `nums[4]`.
Accessing `A(8)` actually accesses `nums[6]`.
Accessing `A(9)` actually accesses `nums[8]`.

Props to apolloydy's solution, I knew the partitioning algorithm already but I didn't know the name. And apolloydy's idea to partition to reverse order happened to make the index rewiring simpler.

written by StefanPochmann original link here

Solution 2

Solution

Roughly speaking I put the smaller half of the numbers on the even indexes and the larger half on the odd indexes.

```
def wiggleSort(self, nums):
    nums.sort()
    half = len(nums[::2])
    nums[::2], nums[1::2] = nums[:half][::-1], nums[half:][::-1]
```

Alternative, maybe nicer, maybe not:

```
def wiggleSort(self, nums):
    nums.sort()
    half = len(nums[::2]) - 1
    nums[::2], nums[1::2] = nums[half::-1], nums[:half:-1]
```

Explanation / Proof

I put the smaller half of the numbers on the even indexes and the larger half on the odd indexes, both from right to left:

```
Example nums = [1,2,...,7]      Example nums = [1,2,...,8]

Small half:  4 . 3 . 2 . 1      Small half:  4 . 3 . 2 . 1 .
Large half:  . 7 . 6 . 5 .      Large half:  . 8 . 7 . 6 . 5
-------------------------       -------------------------
Together:    4 7 3 6 2 5 1      Together:    4 8 3 7 2 6 1 5
```

I want:

- Odd-index numbers are larger than their neighbors.

Since I put the larger numbers on the odd indexes, clearly I already have:

- Odd-index numbers are larger than **or equal to** their neighbors.

Could they be "equal to"? That would require some number M to appear both in the smaller and the larger half. It would be the largest in the smaller half and the smallest in the larger half. Examples again, where S means some number smaller than M and L means some number larger than M.

```
Small half:  M . S . S . S      Small half:  M . S . S . S .
Large half:  . L . L . M .      Large half:  . L . L . L . M
-------------------------       -------------------------
Together:    M L S L S M S      Together:    M L S L S L S M
```

You can see the two M are quite far apart. Of course M could appear more than just

twice, for example:

```
Small half:  M . M . S . S      Small half:  M . S . S . S .
Large half:  . L . L . M .      Large half:  . L . M . M . M
-------------------------      -------------------------
Together:    M L M L S M S      Together:    M L S M S M S M
```

You can see that with seven numbers, three M are no problem. And with eight numbers, four M are no problem. Should be easy to see that in general, with n numbers, floor(n/2) times M is no problem. Now, if there were more M than that, then my method would fail. But... it would also be impossible:

- If n is even, then having more than n/2 times the same number clearly is unsolvable, because you'd have to put two of them next to each other, no matter how you arrange them.
- If n is odd, then the only way to successfully arrange a number appearing more than floor(n/2) times is if it appears exactly floor(n/2)+1 times and you put them on all the even indexes. And to have the wiggle-property, all the other numbers would have to be larger. But then we wouldn't have an M in both the smaller and the larger half.

So if the input has a valid answer at all, then my code will find one.

written by StefanPochmann original link here

Solution 3

```
void wiggleSort(vector<int>& nums) {
    vector<int> sorted(nums);
    sort(sorted.begin(), sorted.end());
    for (int i=nums.size()-1, j=0, k=i/2+1; i>=0; i--)
        nums[i] = sorted[i&1 ? k++ : j++];
}
```

Sort and then write the smaller half of the numbers on the even indexes and the larger half of the numbers on the odd indexes, both from the back. Example:

```
Small half:    4 . 3 . 2 . 1 . 0 .
Large half:    . 9 . 8 . 7 . 6 . 5
--------------------------------
Together:      4 9 3 8 2 7 1 6 0 5
```

So write `nums` from the back, interweaving `sorted[0..4]` (indexed by `j`) and `sorted[5..9]` (indexed by `k`).

For more explanation/proof, see my equivalent Python solution.

written by StefanPochmann original link here

From Leetcoder.

Maximum Size Subarray Sum Equals k

Given an array *nums* and a target value *k*, find the maximum length of a subarray that sums to *k*. If there isn't one, return 0 instead.

Example 1:

Given *nums* = `[1, -1, 5, -2, 3]`, *k* = `3`,
return `4`. (because the subarray `[1, -1, 5, -2]` sums to 3 and is the longest)

Example 2:

Given *nums* = `[-2, -1, 2, 1]`, *k* = `1`,
return `2`. (because the subarray `[-1, 2]` sums to 1 and is the longest)

Follow Up:
Can you do it in O(*n*) time?

Solution 1

```
public int maxSubArrayLen(int[] nums, int k) {
    int sum = 0, max = 0;
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for (int i = 0; i < nums.length; i++) {
        sum = sum + nums[i];
        if (sum == k) max = i + 1;
        else if (map.containsKey(sum - k)) max = Math.max(max, i - map.get(sum -
k));
        if (!map.containsKey(sum)) map.put(sum, i);
    }
    return max;
}
```

The HashMap stores the sum of all elements before index i as key, and i as value. For each i, check not only the current sum but also (currentSum - previousSum) to see if there is any that equals k, and update max length.

PS: An "else" is added. Thanks to beckychiu1988 for comment.

written by he17 original link here

Solution 2

```
class Solution {
public:
    int maxSubArrayLen(vector<int>& nums, int k) {
        unordered_map<int, int> sums;
        int cur_sum = 0;
        int max_len = 0;
        for (int i = 0; i < nums.size(); i++) {
            cur_sum += nums[i];
            if (cur_sum == k) {
                max_len = i + 1;
            } else if (sums.find(cur_sum - k) != sums.end()) {
                max_len = max(max_len, i - sums[cur_sum - k]);
            }
            if (sums.find(cur_sum) == sums.end()) {
                sums[cur_sum] = i;
            }
        }
        return max_len;
    }
};
```

written by RayZ_O original link here

Solution 3

The subarray sum reminds me the range sum problem. Preprocess the input array such that you get the range sum in constant time. sum[i] means the sum from 0 to i inclusively the sum from i to j is sum[j] - sum[i - 1] except that from 0 to j is sum[j].

j-i is equal to the length of subarray of original array. we want to find the max(j - i) for any sum[j] we need to find if there is a previous sum[i] such that sum[j] - sum[i] = k Instead of scanning from 0 to j -1 to find such i, we use hashmap to do the job in constant time. However, there might be duplicate value of of sum[i] we should avoid overriding its index as we want the max j - i, so we want to keep i as left as possible.

```
public class Solution {
    public int maxSubArrayLen(int[] nums, int k) {
        if (nums == null || nums.length == 0)
            return 0;
        int n = nums.length;
        for (int i = 1; i < n; i++)
            nums[i] += nums[i - 1];
        Map<Integer, Integer> map = new HashMap<>();
        map.put(0, -1); // add this fake entry to make sum from 0 to j consistent
        int max = 0;
        for (int i = 0; i < n; i++) {
            if (map.containsKey(nums[i] - k))
                max = Math.max(max, i - map.get(nums[i] - k));
            if (!map.containsKey(nums[i])) // keep only 1st duplicate as we want
first index as left as possible
                map.put(nums[i], i);
        }
        return max;
    }
}
```

written by xuyirui original link here

From Leetcoder.

Power of Three

Given an integer, write a function to determine if it is a power of three.

Follow up:
Could you do it without using any loop / recursion?

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

Well, this problem doesn't seem to be quite interesting or worthwhile to think about at a first glance. I had the same feeling at the beginning. However, after seeing a couple of posts, I saw a couple of interesting ways. So here is a summary post and hope you learn something from others' solutions.

Two trivial solutions first:

Recursive Solution

```
public boolean isPowerOfThree(int n) {
    return n>0 && (n==1 || (n%3==0 && isPowerOfThree(n/3)));
}
```

Iterative Solution

update following Stefan's answer below:

```
public boolean isPowerOfThree(int n) {
    if(n>1)
        while(n%3==0) n /= 3;
    return n==1;
}
```

my original code: public boolean isPowerOfThree(int n) { while(n>1) { if(n%3!=0) return false; n /= 3; } return n<=0 ? false : true; }

It's all about MATH...

Method 1

Find the maximum integer that is a power of 3 and check if it is a multiple of the given input. (related post)

```
public boolean isPowerOfThree(int n) {
    int maxPowerOfThree = (int)Math.pow(3, (int)(Math.log(0x7fffffff) / Math.log(
3)));
    return n>0 && maxPowerOfThree%n==0;
}
```

Or simply hard code it since we know `maxPowerOfThree = 1162261467`:

```
public boolean isPowerOfThree(int n) {
    return n > 0 && (1162261467 % n == 0);
}
```

It is worthwhile to mention that Method 1 works only when the base is prime. For example, we cannot use this algorithm to check if a number is a power of 4 or 6 or any other composite number.

Method 2

If `log10(n) / log10(3)` returns an int (more precisely, a double but has 0 after decimal point), then n is a power of 3. (original post). But **be careful here**, you cannot use `log` (natural log) here, because it will generate round off error for `n=243`. This is more like a coincidence. I mean when `n=243`, we have the following results:

```
log(243) = 5.493061443340548    log(3) = 1.0986122886681098
   ==> log(243)/log(3) = 4.999999999999999

log10(243) = 2.385606273598312    log10(3) = 0.47712125471966244
   ==> log10(243)/log10(3) = 5.0
```

This happens because `log(3)` is actually slightly larger than its true value due to round off, which makes the ratio smaller.

```
public boolean isPowerOfThree(int n) {
    return (Math.log10(n) / Math.log10(3)) % 1 == 0;
}
```

Method 3 related post

```
public boolean isPowerOfThree(int n) {
    return n==0 ? false : n==Math.pow(3, Math.round(Math.log(n) / Math.log(3)));
}
```

Method 4 related post

```
public boolean isPowerOfThree(int n) {
    return n>0 && Math.abs(Math.log10(n)/Math.log10(3)-Math.ceil(Math.log10(n)/Ma
th.log10(3))) < Double.MIN_VALUE;
}
```

`Cheating` Method

This is not really a good idea in general. But for such kind of `power` questions, if we need to check many times, it might be a good idea to store the desired powers into an array first. (related post)

```
public boolean isPowerOfThree(int n) {
    int[] allPowerOfThree = new int[]{1, 3, 9, 27, 81, 243, 729, 2187, 6561, 1968
3, 59049, 177147, 531441, 1594323, 4782969, 14348907, 43046721, 129140163, 387420
489, 1162261467};
    return Arrays.binarySearch(allPowerOfThree, n) >= 0;
}
```

or even better with HashSet:

```
public boolean isPowerOfThree(int n) {
    HashSet<Integer> set = new HashSet<>(Arrays.asList(1, 3, 9, 27, 81, 243, 729,
2187, 6561, 19683, 59049, 177147, 531441, 1594323, 4782969, 14348907, 43046721, 1
29140163, 387420489, 1162261467));
    return set.contains(n);
}
```

New Method Included at 15:30pm Jan-8th

Radix-3 original post

The idea is to convert the original number into radix-3 format and check if it is of format `10*` where `0*` means `k` zeros with `k>=0`.

```
public boolean isPowerOfThree(int n) {
    return Integer.toString(n, 3).matches("10*");
}
```

Any other interesting solutions?

written by GWTW original link here

Solution 2

```
public class Solution {
public bool IsPowerOfThree(int n) {
     return n > 0 && (1162261467 % n == 0);
  }
}
```

written by andrei3 original link here

Solution 3

If `N` is a power of `3`:

- It follows that `3^X == N`
- It follows that `log (3^X) == log N`
- It follows that `X log 3 == log N`
- It follows that `X == (log N) / (log 3)`
- For the basis to hold, `X` must be an integer.

However, due to precision issues that arise from the fact that log 3 cannot be precisely represented on a binary computer; X is considered to be an integer if it's decimal component falls within a guard range of `+/-0.00000000000001`.

```
public boolean isPowerOfThree(int n) {
  double a = Math.log(n) / Math.log(3);
  return Math.abs(a - Math.rint(a)) <= 0.00000000000001;
}
```

written by oluwasayo original link here

From Leetcoder.

Count of Range Sum

Given an integer array `nums`, return the number of range sums that lie in `[lower, upper]` inclusive.
Range sum `S(i, j)` is defined as the sum of the elements in `nums` between indices `i` and `j` (`i` ≤ `j`), inclusive.

Note:
A naive algorithm of $O(n^2)$ is trivial. You MUST do better than that.

Example:
Given *nums* = `[-2, 5, -1]`, *lower* = `-2`, *upper* = `2`,
Return `3`.
The three ranges are : `[0, 0]`, `[2, 2]`, `[0, 2]` and their respective sums are: `-2, -1, 2`.

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

See here for better view

First of all, let's look at the naive solution. Preprocess to calculate the prefix sums `S[i]` = `S(0, i)`, then `S(i, j)` = `S[j] - S[i]`. Note that here we define `S(i, j)` as the sum of range `[i, j)` where `j` exclusive and `j` > `i`. With these prefix sums, it is trivial to see that with `O(n^2)` time we can find all `S(i, j)` in the range `[lower, upper]`

Java - Naive Solution

```
public int countRangeSum(int[] nums, int lower, int upper) {
    int n = nums.length;
    long[] sums = new long[n + 1];
    for (int i = 0; i < n; ++i)
        sums[i + 1] = sums[i] + nums[i];
    int ans = 0;
    for (int i = 0; i < n; ++i)
        for (int j = i + 1; j <= n; ++j)
            if (sums[j] - sums[i] >= lower && sums[j] - sums[i] <= upper)
                ans++;
    return ans;
}
```

However the naive solution is set to TLE intentionally

Now let's do better than this.

Recall count smaller number after self where we encountered the problem

- `count[i]` = count of `nums[j]` - `nums[i]` < `0` with `j` > `i`

Here, after we did the preprocess, we need to solve the problem

- `count[i]` = count of `a` <= `S[j]` - `S[i]` <= `b` with `j` > `i`
- `ans` = sum(count[:])

Therefore the two problems are almost the same. We can use the same technique used in that problem to solve this problem. One solution is merge sort based; another one is Balanced BST based. The time complexity are both `O(n log n)`.

The merge sort based solution counts the answer while doing the merge. During the merge stage, we have already sorted the left half `[start, mid)` and right half `[mid, end)`. We then iterate through the left half with index `i`. For each `i`, we need to find two indices `k` and `j` in the right half where

- `j` is the first index satisfy `sums[j] - sums[i] > upper` and
- `k` is the first index satisfy `sums[k] - sums[i] >= lower`.

Then the number of sums in `[lower, upper]` is `j-k`. We also use another index `t` to copy the elements satisfy `sums[t] < sums[i]` to a cache in order to complete the merge sort.

Despite the nested loops, the time complexity of the "merge & count" stage is still linear. Because the indices `k`, `j`, `t` will only increase but not decrease, each of them will only traversal the right half once at most. The total time complexity of this divide and conquer solution is then `O(n log n)`.

One other concern is that the `sums` may overflow integer. So we use long instead.

Java - Merge Sort Solution

```
public int countRangeSum(int[] nums, int lower, int upper) {
    int n = nums.length;
    long[] sums = new long[n + 1];
    for (int i = 0; i < n; ++i)
        sums[i + 1] = sums[i] + nums[i];
    return countWhileMergeSort(sums, 0, n + 1, lower, upper);
}

private int countWhileMergeSort(long[] sums, int start, int end, int lower, int up
per) {
    if (end - start <= 1) return 0;
    int mid = (start + end) / 2;
    int count = countWhileMergeSort(sums, start, mid, lower, upper)
              + countWhileMergeSort(sums, mid, end, lower, upper);
    int j = mid, k = mid, t = mid;
    long[] cache = new long[end - start];
    for (int i = start, r = 0; i < mid; ++i, ++r) {
        while (k < end && sums[k] - sums[i] < lower) k++;
        while (j < end && sums[j] - sums[i] <= upper) j++;
        while (t < end && sums[t] < sums[i]) cache[r++] = sums[t++];
        cache[r] = sums[i];
        count += j - k;
    }
    System.arraycopy(cache, 0, sums, start, t - start);
    return count;
}
```

written by dietpepsi original link here

Solution 2

Thanks for those contributing excellent ideas to this problem. Here is a quick summary of solutions based on either divide and conquer or binary indexed tree.

To start, we already know there is a straightforward solution by computing each range sum and checking whether it lies in [lower, upper] or not. If the number of elements is n, we have n*(n+1)/2 such range sums so the naive solution will end up with O(n^2) time complexity. Now we are asked to do better than that. So what are the targeted time complexities in your mind? When I first looked at the problem, my instinct is that O(n) solution is too ambitious, so I will target at linearithmic-like (O(n*(logn)^b)) solutions. To get the logarithmic part, it's natural to think of breaking down the original array, and that's where the divide-and-conquer idea comes from.

For this problem, we need some array to apply our divide and conquer algorithm. Without much thinking, we can do that directly with the input array (nums) itself. Since our problem also involves range sums and I believe you have the experience of computing range sums from prefix array of the input array, we might as well apply divide and conquer ideas on the prefix array. So I will give both the input-array based and prefix-array based divide&conquer solutions.

Let's first look at input-array based divide&conquer solution. Our original problem is like this: given an input array nums with length n and a range [lower, upper], find the total number of range sums that lie in the given range. Note the range [lower, upper] and the input array are both fixed. Therefore each range sum can be characterized by two indices i1 and i2 (i1 <= i2), such that range sum S(i1, i2) is the summation of input elements with indices going from i1 up to i2 (both inclusive). Then our problem can be redefined in terms of the value ranges of i1 and i2. For example our original problem can be restated as finding the total number of range sums lying in the given range with 0 <= i1 <= i2 <= n - 1, or in a symbolic way T(0, n-1).

Now if we break our original input array into two subarrays, [0, m] and [m+1, n-1] with m = (n-1)/2, our original problem can be divided into three parts, depending on the values of i1 and i2. If i1 and i2 are both from the first subarray [0, m], we have a subproblem T(0, m); if i1 and i2 are both from the second subarray, we have a subproblem T(m+1, n-1); if i1 is from the first subarray and i2 from the second (note we assume i1 <= i2, therefore we don't have the other case with i2 from first subarray and i1 from second), then we have a new problem which I define as C. In summary we should have:

T(0, n-1) = T(0, m) + T(m+1, n-1) + C

Now from the master theorem, the time complexity of the new problem C should be better than O(n^2), otherwise we make no improvement by applying this divide&conquer idea. So again, I will aim at linearithmic-like solutions for the new problem C: find the total number of range sums lying in the given range with each range sum starting from the first subarray and ending at the second subarray.

First let's try to compute all such range sums. The way I did it was first computing the prefix array of the second subarray and the suffix array (or "backward" prefix array if you like) of the first subarray. Then I can naively add each element in the suffix array to all elements in the prefix array to obtain all the possible range sums. Of course you end up with O(n^2) solution, as expected. So how can we approach it with better time complexity?

Here are the facts I observed: for each element e in the suffix array, we need to add it to all elements in the prefix array. But the order in which we add it doesn't matter. This implies that we can sort our prefix array. This can be done in O(nlogn) time. Now we have a sorted prefix array, do we still need to add the element e to all elements in the prefix array? The answer is no. Because our final goal is to compare the resulted range sums with the given range bounds lower and upper. It is equivalent to modifying the range bounds so we have new bounds (lower - e) and (upper - e) and leave the prefix array unchanged. Now we can compare these new bounds with the sorted prefix array, and I'm sure you can write your own binary search algorithm to do that. So for each element e in the suffix array, we can compute the modified range bounds and get the number of range sums in this new range in logn time. Therefore the total time will be O(nlogn). So in summary, our new problem C can be solved in O(nlogn) time and according to the master theorem, our original problem can be solved in O(n(logn)^2) time. The following is the complete java program:

```
public int countRangeSum(int[] nums, int lower, int upper) {
    if (nums == null || nums.length == 0 || lower > upper) {
        return 0;
    }

    return countRangeSumSub(nums, 0, nums.length - 1, lower, upper);
}

private int countRangeSumSub(int[] nums, int l, int r, int lower, int upper) {
    if (l == r) {
        return nums[l] >= lower && nums[r] <= upper ? 1 : 0;  // base case
    }

    int m = l + (r - l) / 2;
    long[] arr = new long[r - m];  // prefix array for the second subarray
    long sum = 0;
    int count = 0;

    for (int i = m + 1; i <= r; i++) {
        sum += nums[i];
        arr[i - (m + 1)] = sum; // compute the prefix array
    }

    Arrays.sort(arr);  // sort the prefix array

    // Here we can compute the suffix array element by element.
    // For each element in the suffix array, we compute the corresponding
    // "insertion" indices of the modified bounds in the sorted prefix array
    // then the number of valid ranges sums will be given by the indices differen
```

```
ce.
    // I modified the bounds to be "double" to avoid duplicate elements.
    sum = 0;
    for (int i = m; i >= l; i--) {
        sum += nums[i];
        count += findIndex(arr, upper - sum + 0.5) - findIndex(arr, lower - sum -
0.5);
    }

    return countRangeSumSub(nums, l, m, lower, upper) + countRangeSumSub(nums, m
+ 1, r, lower, upper) + count;
}

// binary search function
private int findIndex(long[] arr, double val) {
    int l = 0, r = arr.length - 1, m = 0;

    while (l <= r) {
        m = l + (r - l) / 2;

        if (arr[m] <= val) {
            l = m + 1;
        } else {
            r = m - 1;
        }
    }

    return l;
}
```

(Next two parts will come in answers due to the limitation of maximum length of characters)

written by fun4LeetCode original link here

Solution 3

Understand my segmentTree implementation is not optimized. Please feel free to give me suggestions.

```
public class Solution {
    class SegmentTreeNode {
        SegmentTreeNode left;
        SegmentTreeNode right;
        int count;
        long min;
        long max;
        public SegmentTreeNode(long min, long max) {
            this.min = min;
            this.max = max;
        }
    }
    private SegmentTreeNode buildSegmentTree(Long[] valArr, int low, int high) {
        if(low > high) return null;
        SegmentTreeNode stn = new SegmentTreeNode(valArr[low], valArr[high]);
        if(low == high) return stn;
        int mid = (low + high)/2;
        stn.left = buildSegmentTree(valArr, low, mid);
        stn.right = buildSegmentTree(valArr, mid+1, high);
        return stn;
    }
    private void updateSegmentTree(SegmentTreeNode stn, Long val) {
        if(stn == null) return;
        if(val >= stn.min && val <= stn.max) {
            stn.count++;
            updateSegmentTree(stn.left, val);
            updateSegmentTree(stn.right, val);
        }
    }
    private int getCount(SegmentTreeNode stn, long min, long max) {
        if(stn == null) return 0;
        if(min > stn.max || max < stn.min) return 0;
        if(min <= stn.min && max >= stn.max) return stn.count;
        return getCount(stn.left, min, max) + getCount(stn.right, min, max);
    }

    public int countRangeSum(int[] nums, int lower, int upper) {

        if(nums == null || nums.length == 0) return 0;
        int ans = 0;
        Set<Long> valSet = new HashSet<Long>();
        long sum = 0;
        for(int i = 0; i < nums.length; i++) {
            sum += (long) nums[i];
            valSet.add(sum);
        }

        Long[] valArr = valSet.toArray(new Long[0]);

        Arrays.sort(valArr);
        SegmentTreeNode root = buildSegmentTree(valArr, 0, valArr.length-1);
```

```
        for(int i = nums.length-1; i >=0; i--) {
            updateSegmentTree(root, sum);
            sum -= (long) nums[i];
            ans += getCount(root, (long)lower+sum, (long)upper+sum);
        }
        return ans;
    }

}
```

written by fangyang original link here

From Leetcoder.

Odd Even Linked List

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

Example:
Given `1->2->3->4->5->NULL`,
return `1->3->5->2->4->NULL`.

Note:
The relative order inside both the even and odd groups should remain as it was in the input.
The first node is considered odd, the second node even and so on ...

Credits:
Special thanks to @aadarshjajodia for adding this problem and creating all test cases.

Solution 1

```
public class Solution {
public ListNode oddEvenList(ListNode head) {
    if (head != null) {

        ListNode odd = head, even = head.next, evenHead = even;

        while (even != null && even.next != null) {
            odd.next = odd.next.next;
            even.next = even.next.next;
            odd = odd.next;
            even = even.next;
        }
        odd.next = evenHead;
    }
    return head;
}}
```

written by harrison6 original link here

Solution 2

```
ListNode* oddEvenList(ListNode* head)
{
    if(!head) return head;
    ListNode *odd=head, *evenhead=head->next, *even = evenhead;
    while(even && even->next)
    {
        odd->next = odd->next->next;
        even->next = even->next->next;
        odd = odd->next;
        even = even->next;
    }
    odd->next = evenhead;
    return head;
}
```

written by csk original link here

Solution 3

```
public ListNode oddEvenList(ListNode head) {
    if(head==null||head.next==null) return head;
    ListNode odd=head,ehead=head.next,even=ehead;
    while(even!=null&&even.next!=null){
        odd.next=even.next;
        odd=odd.next;
        even.next=odd.next;
        even=even.next;
    }
    odd.next=ehead;
    return head;
}
```

written by yangneu2015 original link here

From Leetcoder.

Longest Increasing Path in a Matrix

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).

Example 1:

```
nums = [
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
```

Return `4`
The longest increasing path is `[1, 2, 6, 9]`.

Example 2:

```
nums = [
  [3,4,5],
  [3,2,6],
  [2,2,1]
]
```

Return `4`
The longest increasing path is `[3, 4, 5, 6]`. Moving diagonally is not allowed.

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

To get max length of increasing sequences:

1. Do `DFS` from every cell
2. Compare every 4 direction and skip cells that are out of boundary or smaller
3. Get matrix `max` from every cell's `max`
4. Use `matrix[x][y] <= matrix[i][j]` so we don't need a `visited[m][n]` array
5. The key is to `cache` the distance because it's highly possible to revisit a cell

Hope it helps!

```
public static final int[][] dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

public int longestIncreasingPath(int[][] matrix) {
    if(matrix.length == 0) return 0;
    int m = matrix.length, n = matrix[0].length;
    int[][] cache = new int[m][n];
    int max = 1;
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            int len = dfs(matrix, i, j, m, n, cache);
            max = Math.max(max, len);
        }
    }
    return max;
}

public int dfs(int[][] matrix, int i, int j, int m, int n, int[][] cache) {
    if(cache[i][j] != 0) return cache[i][j];
    int max = 1;
    for(int[] dir: dirs) {
        int x = i + dir[0], y = j + dir[1];
        if(x < 0 || x >= m || y < 0 || y >= n || matrix[x][y] <= matrix[i][j]) co
ntinue;
        int len = 1 + dfs(matrix, x, y, m, n, cache);
        max = Math.max(max, len);
    }
    cache[i][j] = max;
    return max;
}
```

written by yavinci original link here

Solution 2

The idea is simple and intuitive:
1. For each cell, try it's left, right, up and down for smaller number.
2. If it's smaller, means we are on the right track and we should keep going. If larger, stop and return.
3. Treat each cell as a start cell. Calculate and memorize the longest distance for this cell, so we don't need to calculate it again in the future.

Questions and advices are welcome.

```
public class Solution {
    public int longestIncreasingPath(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }
        int[][] cache = new int[matrix.length][matrix[0].length];
        int max = 0;
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[0].length; j++) {
                int length = findSmallAround(i, j, matrix, cache, Integer.MAX_VAL
UE);
                max = Math.max(length, max);
            }
        }
        return max;
    }
    private int findSmallAround(int i, int j, int[][] matrix, int[][] cache, int
pre) {
        // if out of bond OR current cell value larger than previous cell value.
        if (i < 0 || i >= matrix.length || j < 0 || j >= matrix[0].length || matr
ix[i][j] >= pre) {
            return 0;
        }
        // if calculated before, no need to do it again
        if (cache[i][j] > 0) {
            return cache[i][j];
        } else {
            int cur = matrix[i][j];
            int tempMax = 0;
            tempMax = Math.max(findSmallAround(i - 1, j, matrix, cache, cur), tem
pMax);
            tempMax = Math.max(findSmallAround(i + 1, j, matrix, cache, cur), tem
pMax);
            tempMax = Math.max(findSmallAround(i, j - 1, matrix, cache, cur), tem
pMax);
            tempMax = Math.max(findSmallAround(i, j + 1, matrix, cache, cur), tem
pMax);
            cache[i][j] = ++tempMax;
            return tempMax;
        }
    }
}
```

written by EdickCoding original link here

Solution 3

We can find longest decreasing path instead, the result will be the same. Use `dp` to record previous results and choose the max `dp` value of smaller neighbors.

```
def longestIncreasingPath(self, matrix):
    def dfs(i, j):
        if not dp[i][j]:
            val = matrix[i][j]
            dp[i][j] = 1 + max(
                dfs(i - 1, j) if i and val > matrix[i - 1][j] else 0,
                dfs(i + 1, j) if i < M - 1 and val > matrix[i + 1][j] else 0,
                dfs(i, j - 1) if j and val > matrix[i][j - 1] else 0,
                dfs(i, j + 1) if j < N - 1 and val > matrix[i][j + 1] else 0)
        return dp[i][j]

    if not matrix or not matrix[0]: return 0
    M, N = len(matrix), len(matrix[0])
    dp = [[0] * N for i in range(M)]
    return max(dfs(x, y) for x in range(M) for y in range(N))
```

written by kitt original link here

From Leetcoder.

Patching Array

Given a sorted positive integer array *nums* and an integer *n*, add/patch elements to the array such that any number in range `[1, n]` inclusive can be formed by the sum of some elements in the array. Return the minimum number of patches required.

Example 1:
nums = `[1, 3]`, *n* = `6`
Return `1`.

Combinations of *nums* are `[1], [3], [1,3]`, which form possible sums of: `1, 3, 4`.
Now if we add/patch `2` to *nums*, the combinations are: `[1], [2], [3], [1,3], [2,3], [1,2,3]`.
Possible sums are `1, 2, 3, 4, 5, 6`, which now covers the range `[1, 6]`.
So we only need `1` patch.

Example 2:
nums = `[1, 5, 10]`, *n* = `20`
Return `2`.
The two patches can be `[2, 4]`.

Example 3:
nums = `[1, 2, 2]`, *n* = `5`
Return `0`.

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

Solution

```
int minPatches(vector<int>& nums, int n) {
    long miss = 1, added = 0, i = 0;
    while (miss <= n) {
        if (i < nums.size() && nums[i] <= miss) {
            miss += nums[i++];
        } else {
            miss += miss;
            added++;
        }
    }
    return added;
}
```

Explanation

Let `miss` be the smallest sum in `[0,n]` that we might be missing. Meaning we already know we can build all sums in `[0,miss)`. Then if we have a number `num <= miss` in the given array, we can add it to those smaller sums to build all sums in `[0,miss+num)`. If we don't, then we must add such a number to the array, and it's best to add `miss` itself, to maximize the reach.

Example: Let's say the input is `nums = [1, 2, 4, 13, 43]` and `n = 100`. We need to ensure that all sums in the range [1,100] are possible.

Using the given numbers 1, 2 and 4, we can already build all sums from 0 to 7, i.e., the range [0,8). But we can't build the sum 8, and the next given number (13) is too large. So we insert 8 into the array. Then we can build all sums in [0,16).

Do we need to insert 16 into the array? No! We can already build the sum 3, and adding the given 13 gives us sum 16. We can also add the 13 to the other sums, extending our range to [0,29).

And so on. The given 43 is too large to help with sum 29, so we must insert 29 into our array. This extends our range to [0,58). But then the 43 becomes useful and expands our range to [0,101). At which point we're done.

Another implementation, though I prefer the above one.

```
int minPatches(vector<int>& nums, int n) {
    int count = 0, i = 0;
    for (long miss=1; miss <= n; count++)
        miss += (i < nums.size() && nums[i] <= miss) ? nums[i++] : miss;
    return count - i;
}
```

written by StefanPochmann original link here

Solution 2

The question asked for the "**minimum** number of patches required". In other words, it asked for an optimal solution. Lots of problems involving optimal solution can be solved by dynamic programming and/or greedy algorithm. I started with greedy algorithm which is conceptually easy to design. Typically, a greedy algorithm needs selection of best moves for a subproblem. So what is our best move?

Think about this example: nums = [1, 2, 3, 9]. We naturally want to iterate through nums from left to right and see what we would discover. After we encountered 1, we know 1...1 is patched completely. After encountered 2, we know 1...3 (1+2) is patched completely. After we encountered 3, we know 1...6 (1+2+3) is patched completely. After we encountered 9, the smallest number we can get is 9. So we must patch a new number here so that we don't miss 7, 8. To have 7, the numbers we can patch is 1, 2, 3 ... 7. Any number greater than 7 won't help here. Patching 8 will not help you get 7. So we have 7 numbers (1...7) to choose from. I hope you can see number 7 works best here because if we chose number 7, we can move all the way up to 1+2+3+7 = 13. (1...13 is patched completely) and it makes us reach n as quickly as possible. After we patched 7 and reach 13, we can consider last element 9 in nums. Having 9 makes us reach 13+9 = 22, which means 1...22 is completely patched. If we still did't reach n, we can then patch 23, which makes 1...45 (22+23) completely patched. We continue until we reach n.

written by bravejia original link here

Solution 3

```
class Solution {
public:
    int minPatches(vector<int>& nums, int n) {
        if (n == 0) return 0;
        int num = nums.size();
        long reach = 0;
        int patch = 0;
        for (int i = 0; i < num; ){
            while (nums[i] > reach + 1){
                reach += (reach + 1);
                ++patch;
                if (reach >= n) return patch;
            }
            reach += nums[i];
            if (reach >= n) return patch;
            ++i;
        }
        while (reach < n){
            reach += (reach + 1);
            ++patch;
        }
        return patch;
    }
};
```

written by AlgoGuruZ original link here

From Leetcoder.

Verify Preorder Serialization of a Binary Tree

One way to serialize a binary tree is to use pre-oder traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as `#`.

```
     _9_
    /   \
   3     2
  / \   / \
 4   1  #  6
/ \ / \   / \
# # # #   # #
```

For example, the above binary tree can be serialized to the string `"9,3,4,#,#,1,#,#,2,#,6,#,#"`, where `#` represents a null node.

Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.

Each comma separated value in the string must be either an integer or a character `'#'` representing `null` pointer.

You may assume that the input format is always valid, for example it could never contain two consecutive commas such as `"1,,3"`.

Example 1:
`"9,3,4,#,#,1,#,#,2,#,6,#,#"`
Return `true`

Example 2:
`"1,#"`
Return `false`

Example 3:
`"9,#,#,1"`
Return `false`

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

Some used stack. Some used the depth of a stack. Here I use a different perspective. In a binary tree, if we consider null as leaves, then

- all non-null node provides 2 outdegree and 1 indegree (2 children and 1 parent), except root
- all null node provides 0 outdegree and 1 indegree (0 child and 1 parent).

Suppose we try to build this tree. During building, we record the difference between out degree and in degree `diff` = `outdegree - indegree`. When the next node comes, we then decrease `diff` by 1, because the node provides an in degree. If the node is not `null`, we increase diff by `2`, because it provides two out degrees. If a serialization is correct, diff should never be negative and diff will be zero when finished.

```
public boolean isValidSerialization(String preorder) {
    String[] nodes = preorder.split(",");
    int diff = 1;
    for (String node: nodes) {
        if (--diff < 0) return false;
        if (!node.equals("#")) diff += 2;
    }
    return diff == 0;
}
```

written by dietpepsi original link here

Solution 2

This is very simple problem if you use stacks. The key here is, when you see two consecutive "#" characters on stack, pop both of them and replace the topmost element on the stack with "#". For example,

preorder = 1,2,3,#,#,#,#

Pass 1: stack = [1]

Pass 2: stack = [1,2]

Pass 3: stack = [1,2,3]

Pass 4: stack = [1,2,3,#]

Pass 5: stack = [1,2,3,#,#] -> two #s on top so pop them and replace top with #. -> stack = [1,2,#]

Pass 6: stack = [1,2,#,#] -> two #s on top so pop them and replace top with #. -> stack = [1,#]

Pass 7: stack = [1,#,#] -> two #s on top so pop them and replace top with #. -> stack = [#]

If there is only one # on stack at the end of the string then return True else return False.

Here is the code for that,

```
class Solution(object):
def isValidSerialization(self, preorder):
    """
    :type preorder: str
    :rtype: bool
    """
    stack = []
    top = -1
    preorder = preorder.split(',')
    for s in preorder:
        stack.append(s)
        top += 1
        while(self.endsWithTwoHashes(stack,top)):
            h = stack.pop()
            top -= 1
            h = stack.pop()
            top -= 1
            if top < 0:
                return False
            h = stack.pop()
            stack.append('#')
        #print stack
    if len(stack) == 1:
        if stack[0] == '#':
            return True
    return False

def endsWithTwoHashes(self,stack,top):
    if top<1:
        return False
    if stack[top]=='#' and stack[top-1]=='#':
        return True
    return False
```

written by harshaneel original link here

Solution 3

We just need to remember how many empty slots we have during the process.

Initially we have one (for the root).

for each node we check if we still have empty slots to put it in.

- a null node occupies one slot.
- a non-null node occupies one slot before he creates two more. the net gain is one.

```
class Solution(object):
    def isValidSerialization(self, preorder):
        """
        :type preorder: str
        :rtype: bool
        """
        # remember how many empty slots we have
        # non-null nodes occupy one slot but create two new slots
        # null nodes occupy one slot

        p = preorder.split(',')

        #initially we have one empty slot to put the root in it
        slot = 1
        for node in p:

            # no empty slot to put the current node
            if slot == 0:
                return False

            # a null node?
            if node == '#':
                # ocuppy slot
                slot -= 1
            else:
                # create new slot
                slot += 1

        #we don't allow empty slots at the end
        return slot==0
```

written by hohomi original link here

From Leetcoder.

Reconstruct Itinerary

Given a list of airline tickets represented by pairs of departure and arrival airports `[from, to]`, reconstruct the itinerary in order. All of the tickets belong to a man who departs from `JFK`. Thus, the itinerary must begin with `JFK`.

Note:

1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary `["JFK", "LGA"]` has a smaller lexical order than `["JFK", "LGB"]`.
2. All airports are represented by three capital letters (IATA code).
3. You may assume all tickets may form at least one valid itinerary.

Example 1:
`tickets` = `[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]`
Return `["JFK", "MUC", "LHR", "SFO", "SJC"]`.

Example 2:
`tickets` = `[["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"], ["ATL","SFO"]]`
Return `["JFK","ATL","JFK","SFO","ATL","SFO"]`.
Another possible reconstruction is `["JFK","SFO","ATL","JFK","ATL","SFO"]`.
But it is larger in lexical order.

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

Just Eulerian path. Greedy DFS, building the route backwards when retreating.

More explanation and example under the codes.

Iterative versions inspired by fangyang (I had only thought of recursion, d'oh).

Ruby

```
def find_itinerary(tickets)
  tickets = tickets.sort.reverse.group_by(&:first)
  route = []
  visit = -> airport {
    visit[tickets[airport].pop()[1]] while (tickets[airport] || []).any?
    route << airport
  }
  visit["JFK"]
  route.reverse
end
```

Iterative version:

```
def find_itinerary(tickets)
  tickets = tickets.sort.reverse.group_by(&:first)
  route, stack = [], ["JFK"]
  while stack.any?
    stack << tickets[stack[-1]].pop()[1] while (tickets[stack[-1]] || []).any?
    route << stack.pop()
  end
  route.reverse
end
```

Python

```
def findItinerary(self, tickets):
    targets = collections.defaultdict(list)
    for a, b in sorted(tickets)[::-1]:
        targets[a] += b,
    route = []
    def visit(airport):
        while targets[airport]:
            visit(targets[airport].pop())
        route.append(airport)
    visit('JFK')
    return route[::-1]
```

Iterative version:

```python
def findItinerary(self, tickets):
    targets = collections.defaultdict(list)
    for a, b in sorted(tickets)[::-1]:
        targets[a] += b,
    route, stack = [], ['JFK']
    while stack:
        while targets[stack[-1]]:
            stack += targets[stack[-1]].pop(),
        route += stack.pop(),
    return route[::-1]
```

Java

```java
public List<String> findItinerary(String[][] tickets) {
    for (String[] ticket : tickets)
        targets.computeIfAbsent(ticket[0], k -> new PriorityQueue()).add(ticket[1
]);
    visit("JFK");
    return route;
}

Map<String, PriorityQueue<String>> targets = new HashMap<>();
List<String> route = new LinkedList();

void visit(String airport) {
    while(targets.containsKey(airport) && !targets.get(airport).isEmpty())
        visit(targets.get(airport).poll());
    route.add(0, airport);
}
```

Iterative version:

```java
public List<String> findItinerary(String[][] tickets) {
    Map<String, PriorityQueue<String>> targets = new HashMap<>();
    for (String[] ticket : tickets)
        targets.computeIfAbsent(ticket[0], k -> new PriorityQueue()).add(ticket[1
]);
    List<String> route = new LinkedList();
    Stack<String> stack = new Stack<>();
    stack.push("JFK");
    while (!stack.empty()) {
        while (targets.containsKey(stack.peek()) && !targets.get(stack.peek()).is
Empty())
            stack.push(targets.get(stack.peek()).poll());
        route.add(0, stack.pop());
    }
    return route;
}
```

C++

```
vector<string> findItinerary(vector<pair<string, string>> tickets) {
    for (auto ticket : tickets)
        targets[ticket.first].insert(ticket.second);
    visit("JFK");
    return vector<string>(route.rbegin(), route.rend());
}

map<string, multiset<string>> targets;
vector<string> route;

void visit(string airport) {
    while (targets[airport].size()) {
        string next = *targets[airport].begin();
        targets[airport].erase(targets[airport].begin());
        visit(next);
    }
    route.push_back(airport);
}
```

Explanation

First keep going forward until you get stuck. That's a good main path already. Remaining tickets form cycles which are found on the way back and get merged into that main path. By writing down the path backwards when retreating from recursion, merging the cycles into the main path is easy - the end part of the path has already been written, the start part of the path hasn't been written yet, so just write down the cycle now and then keep backwards-writing the path.

Example:

From JFK we first visit JFK -> A -> C -> D -> A. There we're stuck, so we write down A as the end of the route and retreat back to D. There we see the unused ticket to B and follow it: D -> B -> C -> JFK -> D. Then we're stuck again, retreat and write down the airports while doing so: Write down D before B, then JFK before D, etc. When we're back from our cycle at D, the written route is D -> B -> C -> JFK -> D -> A. Then we retreat further along the original path, prepending C, A and finally JFK to the route, ending up with the route JFK -> A -> C -> D -> B -> C -> JFK -> D ->

A.

written by StefanPochmann original link here

Solution 2

```
class Solution {
public:
    vector<string> findItinerary(vector<pair<string, string>> tickets) {
        // Each node (airport) contains a set of outgoing edges (destination).
        unordered_map<string, multiset<string>> graph;
        // We are always appending the deepest node to the itinerary,
        // so will need to reverse the itinerary in the end.
        vector<string> itinerary;
        if (tickets.size() == 0){
            return itinerary;
        }
        // Construct the node and assign outgoing edges
        for (pair<string, string> eachTicket : tickets){
            graph[eachTicket.first].insert(eachTicket.second);
        }
        stack<string> dfs;
        dfs.push("JFK");
        while (!dfs.empty()){
            string topAirport = dfs.top();
            if (graph[topAirport].empty()){
                // If there is no more outgoing edges, append to itinerary
                // Two cases:
                // 1. If it searchs the terminal end first, it will simply get
                //    added to the itinerary first as it should, and the proper r
oute
                //    will still be traversed since its entry is still on the sta
ck.
                // 2. If it search the proper route first, the dead end route wil
l also
                //    get added to the itinerary first.
                itinerary.push_back(topAirport);
                dfs.pop();
            }
            else {
                // Otherwise push the outgoing edge to the dfs stack and
                // remove it from the node.
                dfs.push(*(graph[topAirport].begin()));
                graph[topAirport].erase(graph[topAirport].begin());
            }
        }
        // Reverse the itinerary.
        reverse(itinerary.begin(), itinerary.end());
        return itinerary;
    }
};
```

written by frederick2 original link here

Solution 3

Noticed some folks are using Hierholzer's algorithm to find a Eulerian path.

My solution is similar, considering this passenger has to be physically in one place before move to another airport, we are considering using up all tickets and choose lexicographically smaller solution if in tie as two constraints.

Thinking as that passenger, the passenger choose his/her flight greedy as the lexicographical order, once he/she figures out go to an airport without departure with more tickets at hand. the passenger will push current ticket in a stack and look at whether it is possible for him/her to travel to other places from the airport on his/her way.

Please let me know if you have any suggestions.

```
    public List<String> findItinerary(String[][] tickets) {
        List<String> ans = new ArrayList<String>();
        if(tickets == null || tickets.length == 0) return ans;
        Map<String, PriorityQueue<String>> ticketsMap = new HashMap<>();
        for(int i = 0; i < tickets.length; i++) {
            if(!ticketsMap.containsKey(tickets[i][0])) ticketsMap.put(tickets[i][
0], new PriorityQueue<String>());
            ticketsMap.get(tickets[i][0]).add(tickets[i][1]);
        }

        String curr = "JFK";
        Stack<String> drawBack = new Stack<String>();
        for(int i = 0; i < tickets.length; i++) {
            while(!ticketsMap.containsKey(curr) || ticketsMap.get(curr).isEmpty()
) {
                drawBack.push(curr);
                curr = ans.remove(ans.size()-1);
            }
            ans.add(curr);
            curr = ticketsMap.get(curr).poll();
        }
        ans.add(curr);
        while(!drawBack.isEmpty()) ans.add(drawBack.pop());
        return ans;
    }
```

written by fangyang original link here

From Leetcoder.

Largest BST Subtree

Given a binary tree, find the largest subtree which is a Binary Search Tree (BST), where largest means subtree with largest number of nodes in it.

Note:
A subtree must include all of its descendants.
Here's an example:

```
     10
     / \
    5  15
   / \   \
  1   8   7
```

The Largest BST Subtree in this case is the highlighted one.
The return value is the subtree's size, which is 3.

1. You can recursively use algorithm similar to 98. Validate Binary Search Tree at each node of the tree, which will result in O(nlogn) time complexity.

Follow up:
Can you figure out ways to solve it with O(n) time complexity?

Solution 1

> edited code: thanks @hyj143 and @petrichory

```
public class Solution {

    class Result {  // (size, rangeLower, rangeUpper) -- size of current tree, ra
nge of current tree [rangeLower, rangeUpper]
        int size;
        int lower;
        int upper;

        Result(int size, int lower, int upper) {
            this.size = size;
            this.lower = lower;
            this.upper = upper;
        }
    }

    int max = 0;

    public int largestBSTSubtree(TreeNode root) {
        if (root == null) { return 0; }
        traverse(root);
        return max;
    }

    private Result traverse(TreeNode root) {
        if (root == null) { return new Result(0, Integer.MAX_VALUE, Integer.MIN_V
ALUE); }
        Result left = traverse(root.left);
        Result right = traverse(root.right);
        if (left.size == -1 || right.size == -1 || root.val <= left.upper || root
.val >= right.lower) {
            return new Result(-1, 0, 0);
        }
        int size = left.size + 1 + right.size;
        max = Math.max(size, max);
        return new Result(size, Math.min(left.lower, root.val), Math.max(right.up
per, root.val));
    }
}
```

```java
/*
    in brute-force solution, we get information in a top-down manner.
    for O(n) solution, we do it in bottom-up manner, meaning we collect informati
on during backtracking.
*/
public class Solution {

    class Result {  // (size, rangeLower, rangeUpper) -- size of current tree, ra
nge of current tree [rangeLower, rangeUpper]
        int size;
        int lower;
        int upper;

        Result(int size, int lower, int upper) {
            this.size = size;
            this.lower = lower;
            this.upper = upper;
        }
    }

    int max = 0;

    public int largestBSTSubtree(TreeNode root) {
        if (root == null) { return 0; }
        traverse(root, null);
        return max;
    }

    private Result traverse(TreeNode root, TreeNode parent) {
        if (root == null) { return new Result(0, parent.val, parent.val); }
        Result left = traverse(root.left, root);
        Result right = traverse(root.right, root);
        if (left.size==-1 || right.size==-1 || root.val<left.upper || root.val>ri
ght.lower) {
            return new Result(-1, 0, 0);
        }
        int size = left.size + 1 + right.size;
        max = Math.max(size, max);
        return new Result(size, left.lower, right.upper);
    }
}
```

written by mach7 original link here

Solution 2

```
def largestBSTSubtree(self, root):
    def dfs(root):
        if not root:
            return 0, 0, float('inf'), float('-inf')
        N1, n1, min1, max1 = dfs(root.left)
        N2, n2, min2, max2 = dfs(root.right)
        n = n1 + 1 + n2 if max1 < root.val < min2 else float('-inf')
        return max(N1, N2, n), n, min(min1, root.val), max(max2, root.val)
    return dfs(root)[0]
```

My `dfs` returns four values:

- `N` is the size of the largest BST in the tree.
- If the tree is a BST, then `n` is the number of nodes, otherwise it's -infinity.
- If the tree is a BST, then `min` and `max` are the minimum/maximum value in the tree.

written by StefanPochmann original link here

Solution 3

```
public int largestBSTSubtree(TreeNode root) {
    if (root == null) return 0;
    if (root.left == null && root.right == null) return 1;
    if (isValid(root, null, null)) return countNode(root);
    return Math.max(largestBSTSubtree(root.left), largestBSTSubtree(root.right));
}

public boolean isValid(TreeNode root, Integer min, Integer max) {
    if (root == null) return true;
    if (min != null && min >= root.val) return false;
    if (max != null && max <= root.val) return false;
    return isValid(root.left, min, root.val) && isValid(root.right, root.val, max)
;
}

public int countNode(TreeNode root) {
    if (root == null) return 0;
    if (root.left == null && root.right == null) return 1;
    return 1 + countNode(root.left) + countNode(root.right);
}
```

written by hollysinka original link here

From Leetcoder.

Increasing Triplet Subsequence

Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.

Formally the function should:

> Return true if there exists i, j, k
> such that $arr[i] < arr[j] < arr[k]$ given $0 \le i < j < k \le n-1$ else return false.

Your algorithm should run in $O(n)$ time complexity and $O(1)$ space complexity.

Examples:
Given `[1, 2, 3, 4, 5]`,
return `true`.

Given `[5, 4, 3, 2, 1]`,
return `false`.

Credits:
Special thanks to @DjangoUnchained for adding this problem and creating all test cases.

Solution 1

```
bool increasingTriplet(vector<int>& nums) {
    int c1 = INT_MAX, c2 = INT_MAX;
    for (int x : nums) {
        if (x <= c1) {
            c1 = x;           // c1 is min seen so far (it's a candidate for 1st
element)
        } else if (x <= c2) { // here when x > c1, i.e. x might be either c2 or c
3
            c2 = x;           // x is better than the current c2, store it
        } else {              // here when we have/had c1 < c2 already and x > c2
            return true;      // the increasing subsequence of 3 elements exists
        }
    }
    return false;
}
```

written by alveko original link here

Solution 2

```
   public boolean increasingTriplet(int[] nums) {
        // start with two largest values, as soon as we find a number bigger than
both, while both have been updated, return true.
        int small = Integer.MAX_VALUE, big = Integer.MAX_VALUE;
        for (int n : nums) {
            if (n <= small) { small = n; } // update small if n is smaller than b
oth
            else if (n <= big) { big = n; } // update big only if greater than sm
all but smaller than big
            else return true; // return if you find a number bigger than both
        }
        return false;
    }
```

written by sim5 original link here

Solution 3

The main idea is keep two values when check all elements in the array: the minimum value *min* until now and the second minimum value *secondMin* from the minimum value's position until now. Then if we can find the third one that larger than those two values at the same time, it must exists the triplet subsequence and return true.

What need to be careful is: we need to include the condition that some value has the same value with minimum number, otherwise this condition will cause the secondMin change its value.

```
public class Solution {
    public boolean increasingTriplet(int[] nums) {
        int min = Integer.MAX_VALUE, secondMin = Integer.MAX_VALUE;
        for(int num : nums){
            if(num <= min) min = num;
            else if(num < secondMin) secondMin = num;
            else if(num > secondMin) return true;
        }
        return false;
    }
}
```

The running time complexity is O(n).

written by billleeforfun original link here

From Leetcoder.

Self Crossing

You are given an array *x* of `n` positive numbers. You start at point `(0,0)` and moves `x[0]` metres to the north, then `x[1]` metres to the west, `x[2]` metres to the south, `x[3]` metres to the east and so on. In other words, after each move your direction changes counter-clockwise.

Write a one-pass algorithm with `O(1)` extra space to determine, if your path crosses itself, or not.

Example 1:
Given *x* = `[2, 1, 1, 2]`
Return `true` (self crossing)
Example 2:
Given *x* = `[1, 2, 3, 4]`
Return `false` (not self crossing)
Example 3:
Given *x* = `[1, 1, 1, 1]`
Return `true` (self crossing)

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

```
// Categorize the self-crossing scenarios, there are 3 of them:
// 1. Fourth line crosses first line and works for fifth line crosses second line
and so on...
// 2. Fifth line meets first line and works for the lines after
// 3. Sixth line crosses first line and works for the lines after
public class Solution {
    public boolean isSelfCrossing(int[] x) {
        int l = x.length;
        if(l <= 3) return false;

        for(int i = 3; i < l; i++){
            if(x[i] >= x[i-2] && x[i-1] <= x[i-3]) return true;  //Fourth line cr
osses first line and onward
            if(i >=4)
            {
                if(x[i-1] == x[i-3] && x[i] + x[i-4] >= x[i-2]) return true; // F
ifth line meets first line and onward
            }
            if(i >=5)
            {
                if(x[i-2] - x[i-4] >= 0 && x[i] >= x[i-2] - x[i-4] && x[i-1] >= x
[i-3] - x[i-5] && x[i-1] <= x[i-3]) return true;  // Sixth line crosses first lin
e and onward
            }
        }
        return false;
    }
}
```

written by KuangYuan original link here

Solution 2

After drawing a few crossing cases ourselves, we can simply find out there are two basic patterns:

- x[i-1]<=x[i-3] && x[i]>=x[i-2] the ending circle line cross the beginning circle line in one circle;
- i>=5 && x[i-1]<=x[i-3] && x[i]>=x[i-2]-x[i-4] the second line of the next circle cross the the beginning of the previous circle between two adjacent circles;

But still that is not over yet, how about some special cases? How about the first line of the next circle and the previous circle? Yeah, the beginning line of the next circle can overlap the the first line of the previous circle - another two adjacent circles case:

- i>=4 && x[i-1]==x[i-3] && x[i]>=x[i-2]-x[i-4]

Quite straightforward. Then we can test our patterns now, however soon we will find out that the second cases is not strong enough to cover all possible situations - the second line of the next circle crossing the previous circle at the its first line

- [3,3,3,2,1,1] is an example here, so x[i-2]>=x[i-4] then must be added to our conditions;
- [3,3,4,4,10,4,4,,3,3] is another typical example for x[i-3]<=x[i-1]+x[i-5] condition, which also should be added to make the constrained conditions stronger;

At last, we make it! Bang! End of story with a very terse, clean and efficient code as follows.

```
//AC - 0ms;
bool isSelfCrossing(int* x, int size)
{
    for(int i = 3; i < size; i++)
    {
        if(x[i]>=x[i-2] && x[i-1]<=x[i-3]) return true;
        if(i>=4 && x[i-1]==x[i-3] && x[i]+x[i-4]>=x[i-2]) return true;
        if(i>=5 && x[i-2]-x[i-4]>=0 && x[i]>=x[i-2]-x[i-4] && x[i-1]>=x[i-3]-x[i-
5] && x[i-1]<=x[i-3]) return true;
    }
    return false;
}
```

written by LHearen original link here

Solution 3

Checking out every six pack.

Solution 1

```
def isSelfCrossing(self, x):
    return any(d >= b > 0 and (a >= c or a >= c-e >= 0 and f >= d-b)
               for a, b, c, d, e, f in ((x[i:i+6] + [0] * 6)[:6]
                                        for i in xrange(len(x))))
```

Solution 2

```
def isSelfCrossing(self, x):
    b = c = d = e = 0
    for a in x:
        if d >= b > 0 and (a >= c or a >= c-e >= 0 and f >= d-b):
            return True
        b, c, d, e, f = a, b, c, d, e
    return False
```

Explanation

```
             b                              b
   +----------------+              +----------------+
   |                |              |                |
   |                |              |                | a
 c |                |            c |                |
   |                | a            |                |    f
   +----------->    |              |                | <----+
             d      |              |                |      | e
                    |              |                |
                                   +------------------------+
                                                  d
```

Draw a line of length `a`. Then draw further lines of lengths `b`, `c`, etc. How does the `a`-line get crossed? From the left by the `d`-line or from the right by the `f`-line, see the above picture. I just encoded the criteria for actually crossing it.

Two details:

- In both cases, `d` needs to be at least `b`. In the first case to cross the `a`-line directly, and in the second case to get behind it so that the `f`-line can cross it. So I factored out `d >= b`.
- The "special case" of the `e`-line stabbing the `a`-line from below is covered because in that case, the `f`-line "crosses" it (note that even if there is no actual `f`-line, my code uses `f = 0` and thus still finds that "crossing").

written by StefanPochmann original link here

Palindrome Pairs

Given a list of unique words. Find all pairs of indices `(i, j)` in the given list, so that the concatenation of the two words, i.e. `words[i] + words[j]` is a palindrome.

Example 1:
Given `words` = `["bat", "tab", "cat"]`
Return `[[0, 1], [1, 0]]`
The palindromes are `["battab", "tabbat"]`

Example 2:
Given `words` = `["abcd", "dcba", "lls", "s", "sssll"]`
Return `[[0, 1], [1, 0], [3, 2], [2, 4]]`
The palindromes are `["dcbaabcd", "abcddcba", "slls", "llssssll"]`

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

```
public List<List<Integer>> palindromePairs(String[] words) {
    List<List<Integer>> pairs = new LinkedList<>();
    if (words == null) return pairs;
    HashMap<String, Integer> map = new HashMap<>();
    for (int i = 0; i < words.length; ++ i) map.put(words[i], i);
    for (int i = 0; i < words.length; ++ i) {
        int l = 0, r = 0;
        while (l <= r) {
            String s = words[i].substring(l, r);
            Integer j = map.get(new StringBuilder(s).reverse().toString());
            if (j != null && i != j && isPalindrome(words[i].substring(l == 0 ? r
: 0, l == 0 ? words[i].length() : l)))
                pairs.add(Arrays.asList(l == 0 ? new Integer[]{i, j} : new Intege
r[]{j, i}));
            if (r < words[i].length()) ++r;
            else ++l;
        }
    }
    return pairs;
}

private boolean isPalindrome(String s) {
    for (int i = 0; i < s.length()/2; ++ i)
        if (s.charAt(i) != s.charAt(s.length()-1-i))
            return false;
    return true;
}
```

written by Nursike original link here

Solution 2

Apparently there is a O(n^2*k) naive solution for this problem, with n the total number of words in the "words" array and k the average length of each word:

For each word, we simply go through the "words" array and check whether the concatenated string is a palindrome or not.

Of course this will result in TLE as expected. To improve the algorithm, we need to reduce the number of words that is needed to check for each word, instead of iterating through the whole array. This prompted me to think if I can extract any useful information out of the process of checking whether the concatenated string is a palindrome, so that it can help eliminate as many words as possible for the rest of the "words" array.

First here is the technique I employed to check palindrome: maintain two pointers, i and j, with i pointing to the beginning of the string, j pointing to the end of the string. Characters pointed by i and j are compared. If at any time the characters pointed by them are not the same, we conclude the string is not a palindrome. Otherwise we move the two pointers towards each other until they meet in the middle and the string is a palindrome.

By examining the above process, I do find something that we may take advantage of to get rid of words that need to be checked. For example, let's say we want to append words to w0, which starts with character 'a'. Then we only need to consider words ending with character 'a', i.e., this will single out all words ending with character 'a'. If the second character of w0 is 'b' for instance, we can further reduce our candidate set to words ending with string "ba", etc. Our naive solution throws all the information away and repeats the comparison, which leads to the undesired O(n^2*k) complexity.

In order to exploit the information gathered so far, we obviously need to restructure all the words in the "words" array. If you are familiar with Trie structure (I believe you are, since LeetCode has problems for it. In case you are not, see Trie), it will come to mind as we need to deal with words with common suffixes. The next step is to design fields for each node in the Trie. There are at least two fields that should be covered for each TrieNode: a TrieNode array denoting the next layer of nodes and a boolean (or integer) to signify the end of a word. So our tentative TrieNode will look like this:

```
class TrieNode {
      TrieNode[] next;
      boolean isWord;
 }
```

One point here is that we assume all the words contain lowercase letters only. This is not specified in the problem statement so you probably need to confirm with the interviewer (here I assume it is the case)

Now we will rearrange each word into this Trie structure: simply starting from the

root of the Trie and the ending character of each word, identify the node at the next layer by indexing into root's "next" array with index given by the difference between the ending character and character 'a'. If the indexed node is null, create a new node. Continue to the next layer and towards the beginning of the word in this manner until we are done with the word, at which point we will label the "isWord" field of the final node as true.

After building up the Trie structure, we can proceed to search for pairs of palindromes for each word in the "words" array. I will use the following example to explain how it works and make possible modifications of the TrieNode we proposed above.

Let's say we have these words: ["ba", "a", "aaa"], the Trie structure will be as follows:

```
        root (f)
           | 'a'
          n1 (t)
    ------------
 'b' |          | 'a'
   n2 (t)    n3 (f)
               | 'a'
             n4 (t)
```

The letter in parentheses indicates the value of "isWord" for each node: f ==> false; t ==> true. The character at each vertical line denotes the index into the "next" array of the corresponding node. For example, for the first vertical line, 'a' means root.next[0] is not null. Similarly 'b' means n1.next[1] is not null, and so on.

Here is the searching process:

1. For word "ba", starting from the first character 'b', index into the root.next array with index ('b' - 'a' = 1). The corresponding node is null, then we know there are no words ending at this character, so the search is terminated;
2. For word "a", again indexing into array root.next at index ('a' - 'a' = 0) will yield node n1, which is not null. We then check the value of n1.isWord. If it is true, then it is possible to obtain a palindrome by appending this word to the one currently being examined (a.k.a. word "a"). Also note that the two words should be different, but the n1.isWord field provides no information about the word itself, which makes it impossible to distinguish the two words. So we need to modify the fields of the TrieNode so we can identify the word it represents. One easy way is to have an integer field to remember the index of the word in the "words" array. For non-word nodes, this integer will take negative values (-1 for example) while for those representing a word, it will be non-negative values. Suppose we have made this modification, then we know the two words are the same, so we discard this pair combination. Since the word "a" has only one letter, it seems we are done with it. Or do we? Not really. What if we have words with suffix "a" ("aaa" in this case)? We need to continue check the rest part of these words (such as "aa" for the word "aaa") and see if the rest forms a palindrome. If it is, then appending this word ("aaa" in this case) to the original

word ("a") will also form a palindrome ("aaaa"). Here I take another strategy: add an integer list to each TrieNode; the list will record the indices of all words satisfying the following two conditions: each word has a suffix represented by the current Trie node; the rest of the word forms a palindrome.

Before I get to the third word "aaa", let me spell out the new TrieNode and the corresponding Trie structure for the above array.

```
class TrieNode {
     TrieNode[] next;
     int index;
     List<Integer> list;

    TrieNode() {
       next = new TrieNode[26];
       index = -1;
       list = new ArrayList<>();
    }
}
```

Trie structure:

```
          root (-1,[1])
            | 'a'
          n1 (1,[1,2])
    ---------------------
'b' |                   | 'a'
  n2 (0,[0])    n3 (-1,[2])
                        | 'a'
                 n4 (2,[2])
```

The first integer in the parentheses is the index of the word in the "words" array (defaulted to -1). The integers in the square bracket are the indices of words satisfying the two conditions mentioned above.

Let's continue with the third word "aaa" with this new structure. Indexing into array root.next at index ('a' - 'a' = 0) will yield node n1 and n1.index = 1 >= 0, which is also different from the index of "aaa" (which is 2). So pair (2,1) is a possible concatenation to form a palindrome. But still we need to check the rest of "aaa" (excluding the string represented by current n1 which is "a" from the beginning of "aaa") to see if it is a palindrome. If it is, then (2,1) is a valid combination. We continue in this fashion until we reach the end of "aaa". Lastly we will check n4.list to see if there are any words satisfying the two conditions specified in step 2 which are different from current word, and add the corresponding valid pairs.

Both building and searching the Trie structure take O(n*k^2), which set the total time complexity of the solution. See the complete Java program in the answer.

written by fun4LeetCode original link here

Solution 3

There is no requirement that states i and j should be different in (i, j) pairs. So in example 2 (3, 3) should lead to 'ss' which is a palindrome. Problem statement should be corrected as:

Find all pairs of ***distinct*** indices (i, j) in the given list, so that

written by smirzai original link here

From Leetcoder.

House Robber III

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

Example 1:

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

Maximum amount of money the thief can rob = 3 + 3 + 1 = **7**.

Example 2:

```
     3
    / \
   4   5
  / \   \
 1   3   1
```

Maximum amount of money the thief can rob = 4 + 5 = **9**.

Credits:

Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

Step I -- Think naively

At first glance, the problem exhibits the feature of "optimal substructure": if we want to "rob" maximum amount of money from current binary tree (rooted at "root"), we surely hope that we can do the same to its left and right subtrees.

So going along this line, let's define the function `rob(root)` which will return the maximum amount of money that we can rob for the binary tree rooted at "root"; the key now is to construct the solution to the original problem from solutions to its subproblems, i.e., how to get `rob(root)` from `rob(root.left), rob(root.right), ...` etc.

Apparently the analyses above suggest a recursive solution. And for recursion, it's always worthwhile to figure out the following two properties:

1. Termination condition: when do we know the answer to `rob(root)` without any calculation? Of course when the tree is empty -- we've got nothing to rob so the amount of money is zero.
2. Recurrence relation: i.e., how to get `rob(root)` from `rob(root.left), rob(root.right), ...` etc. From the point of view of the tree root, there are only two scenarios at the end: "root" is robbed or is not. If it is, due to the constraint that "we cannot rob any two directly-linked houses", the next level of subtrees that are available would be the four "grandchild-subtrees" (*root.left.left, root.left.right, root.right.left, root.right.right*). However if root is not robbed, the next level of available subtrees would just be the two "child-subtrees" (*root.left, root.right*). We only need to choose the scenario which yields the larger amount of money.

Here is the program for the ideas above:

```
public int rob(TreeNode root) {
    if (root == null) {
        return 0;
    }

    int val = 0;

    if (root.left != null) {
        val += rob(root.left.left) + rob(root.left.right);
    }

    if (root.right != null) {
        val += rob(root.right.left) + rob(root.right.right);
    }

    return Math.max(val + root.val, rob(root.left) + rob(root.right));
}
```

However the solution runs very slow (1186 ms) and barely got accepted.

Step II -- Think one step further

In step I, we only considered the aspect of "optimal substructure", but think little about the possibilities of overlapping of the subproblems. For example, to obtain `rob(root)`, we need `rob(root.left), rob(root.right), rob(root.left.left), rob(root.left.right), rob(root.right.left), rob(root.right.right)`; but to get `rob(root.left)`, we also need `rob(root.left.left), rob(root.left.right)`, similarly for `rob(root.right)`. The naive solution above computed these subproblems repeatedly, which resulted in bad time performance. Now if you recall the two conditions for dynamic programming: "**optimal substructure**" + "**overlapping of subproblems**", we actually have a DP problem. A naive way to implement DP here is to use a hash map to record the results for visited subtrees.

And here is the improved solution:

```
public int rob(TreeNode root) {
    Map<TreeNode, Integer> map = new HashMap<>();
    return robSub(root, map);
}

private int robSub(TreeNode root, Map<TreeNode, Integer> map) {
    if (root == null) return 0;
    if (map.containsKey(root)) return map.get(root);

    int val = 0;

    if (root.left != null) {
        val += robSub(root.left.left, map) + robSub(root.left.right, map);
    }

    if (root.right != null) {
        val += robSub(root.right.left, map) + robSub(root.right.right, map);
    }

    val = Math.max(val + root.val, robSub(root.left, map) + robSub(root.right, ma
p));
    map.put(root, val);

    return val;
}
```

The runtime is sharply reduced to 9ms, at the expense of O(n) space cost (n is the total number of nodes; stack cost for recursion is not counted).

Step III -- Think one step back

In step I, we defined our problem as `rob(root)`, which will yield the maximum amount of money that can be robbed of the binary tree rooted at "root". This leads to the DP problem summarized in step II.

Now let's take one step back and ask why do we have overlapping subproblems? If you trace all the way back to the beginning, you'll find the answer lies in the way

how we have defined `rob(root)`. As I mentioned, for each tree root, there are two scenarios: it is robbed or is not. `rob(root)` does not distinguish between these two cases, so "information is lost as the recursion goes deeper and deeper", which resulted in repeated subproblems.

If we were able to maintain the information about the two scenarios for each tree root, let's see how it plays out. Redefine `rob(root)` as a new function which will return an array of two elements, the first element of which denotes the maximum amount of money that can be robbed if "root" is **not robbed**, while the second element signifies the maximum amount of money robbed if root is **robbed**.

Let's relate `rob(root)` to `rob(root.left)` and `rob(root.right)`, etc. For the 1st element of `rob(root)`, we only need to sum up the larger elements of `rob(root.left)` and `rob(root.right)`, respectively, since root is not robbed and we are free to rob the left and right subtrees. For the 2nd element of `rob(root)`, however, we only need to add up the 1st elements of `rob(root.left)` and `rob(root.right)`, respectively, plus the value robbed from "root" itself, since in this case it's guaranteed that we cannot rob the nodes of root.left and root.right.

As you can see, by keeping track of the information of both scenarios, we decoupled the subproblems and the solution essentially boiled down to a greedy one. Here is the program:

```
public int rob(TreeNode root) {
    int[] res = robSub(root);
    return Math.max(res[0], res[1]);
}

private int[] robSub(TreeNode root) {
    if (root == null) {
        return new int[2];
    }

    int[] left = robSub(root.left);
    int[] right = robSub(root.right);

    int[] res = new int[2];
    res[0] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
    res[1] = root.val + left[0] + right[0];

    return res;
}
```

written by fun4LeetCode original link here

Solution 2

dfs all the nodes of the tree, each node return two number, int[] num, num[0] is the max value while rob this node, num[1] is max value while not rob this value. Current node return value only depend on its children's value. Transform function should be very easy to understand.

```
public class Solution {
    public int rob(TreeNode root) {
        int[] num = dfs(root);
        return Math.max(num[0], num[1]);
    }
    private int[] dfs(TreeNode x) {
        if (x == null) return new int[2];
        int[] left = dfs(x.left);
        int[] right = dfs(x.right);
        int[] res = new int[2];
        res[0] = left[1] + right[1] + x.val;
        res[1] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
        return res;
    }
}
```

written by jiangbowei2010 original link here

Solution 3

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int tryRob(TreeNode* root, int& l, int& r) {
        if (!root)
            return 0;

        int ll = 0, lr = 0, rl = 0, rr = 0;
        l = tryRob(root->left, ll, lr);
        r = tryRob(root->right, rl, rr);

        return max(root->val + ll + lr + rl + rr, l + r);
    }

    int rob(TreeNode* root) {
        int l, r;
        return tryRob(root, l, r);
    }
};
```

Basically you want to compare which one is bigger between 1) you + sum of your grandchildren and 2) sum of your children. Personally I like my solution better than the most voted solution because I don't need complex data structures like map.

written by espuer original link here

From Leetcoder.

Counting Bits

Given a non negative integer number **num**. For every numbers **i** in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

Example: For `num = 5` you should return `[0,1,1,2,1,2]`.

Follow up:

- It is very easy to come up with a solution with run time **O(n*sizeof(integer))**. But can you do it in linear time**O(n)** /possibly in a single pass?
- Space complexity should be **O(n)**.
- Can you do it like a boss? Do it without using any builtin function like **__builtin_popcount** in c++ or in any other language.

1. You should make use of what you have produced already.
2. Divide the numbers in ranges like [2-3], [4-7], [8-15] and so on. And try to generate new range from previous.
3. Or does the odd/even status of the number help you in calculating the number of 1s?

Credits:
Special thanks to @ syedee for adding this problem and creating all test cases.

Solution 1

An easy recurrence for this problem is f[i] = f[i / 2] + i % 2.

```
public int[] countBits(int num) {
    int[] f = new int[num + 1];
    for (int i=1; i<=num; i++) f[i] = f[i >> 1] + (i & 1);
    return f;
}
```

written by lixx2100 original link here

Solution 2

```
class Solution {
public:
    vector<int> countBits(int num) {
        vector<int> ret(num+1, 0);
        for (int i = 1; i <= num; ++i)
            ret[i] = ret[i&(i-1)] + 1;
        return ret;
    }
};
```

written by fengcc original link here

Solution 3

```
class Solution {
public:
    vector<int> countBits(int num) {

        vector<int> res(num+1,0);

        for(int i = 1; i < res.size();i++)
            res[i] = i%2 + res[i/2];

        return res;
    }
};
```

written by m.shyamkrishnan original link here

From Leetcoder.

Nested List Weight Sum

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1:
Given the list `[[1,1],2,[1,1]]`, return **10**. (four 1's at depth 2, one 2 at depth 1)

Example 2:
Given the list `[1,[4,[6]]]`, return **27**. (one 1 at depth 1, one 4 at depth 2, and one 6 at depth 3; 1 + 4*2 + 6*3 = 27)

Solution 1

```
public int depthSum(List<NestedInteger> nestedList) {
    return helper(nestedList, 1);
}

private int helper(List<NestedInteger> list, int depth)
{
    int ret = 0;
    for (NestedInteger e: list)
    {
        ret += e.isInteger()? e.getInteger() * depth: helper(e.getList(), depth +
1);
    }
    return ret;
}
```

written by larrywang2014 original link here

Solution 2

```
public int depthSum(List<NestedInteger> nestedList) {
    if(nestedList == null){
        return 0;
    }

    int sum = 0;
    int level = 1;

    Queue<NestedInteger> queue = new LinkedList<NestedInteger>(nestedList);
    while(queue.size() > 0){
        int size = queue.size();

        for(int i = 0; i < size; i++){
            NestedInteger ni = queue.poll();

            if(ni.isInteger()){
                sum += ni.getInteger() * level;
            }else{
                queue.addAll(ni.getList());
            }
        }

        level++;
    }

    return sum;
}
```

written by lop original link here

Solution 3

c++:

```
class Solution {
    private:
        int DFS(vector<NestedInteger>& nestedList, int depth){
            int n = (int)nestedList.size();
            int sum = 0;
            for(int i=0;i<n;i++){
                if(nestedList[i].isInteger()){
                    sum += nestedList[i].getInteger()*depth;
                }
                else{
                    sum += DFS(nestedList[i].getList(),depth+1);
                }
            }
            return sum;
        }
    public:
        int depthSum(vector<NestedInteger>& nestedList) {
            return DFS(nestedList, 1);
        }
};
```

Python:

```
class Solution(object):
    def depthSum(self, nestedList):
        """
        :type nestedList: List[NestedInteger]
        :rtype: int
        """
        def DFS(nestedList, depth):
            temp_sum = 0
            for member in nestedList:
                if member.isInteger():
                    temp_sum += member.getInteger() * depth
                else:
                    temp_sum += DFS(member.getList(),depth+1)
            return temp_sum
        return DFS(nestedList,1)
```

Javascript:

```
/**
 * @param {NestedInteger[]} nestedList
 * @return {number}
 */
var dfs = function(nestedList,depth){
    var sum = 0;
    var n = nestedList.length;
    for(var i=0; i<n;i++){
        if(nestedList[i].isInteger()){
            sum += nestedList[i].getInteger() * depth;
        }
        else{
            sum += dfs(nestedList[i].getList(),depth+1);
        }
    }
    return sum;
};
var depthSum = function(nestedList) {
  return dfs(nestedList,1);
};
```

written by sxycwzwzq original link here

From Leetcoder.

Longest Substring with At Most K Distinct Characters

Given a string, find the length of the longest substring T that contains at most *k* distinct characters.

For example, Given s = `“eceba”` and k = 2,

T is "ece" which its length is 3.

Solution 1

feel it is not a new question, just use num to track the number of distinct characters within the slide window

```
public class Solution {
    public int lengthOfLongestSubstringKDistinct(String s, int k) {
        int[] count = new int[256];
        int num = 0, i = 0, res = 0;
        for (int j = 0; j < s.length(); j++) {
            if (count[s.charAt(j)]++ == 0) num++;
            if (num > k) {
                while (--count[s.charAt(i++)] > 0);
                num--;
            }
            res = Math.max(res, j - i + 1);
        }
        return res;
    }
}
```

written by jiangbowei2010 original link here

Solution 2

da jia zhao gong zuo jia you!!!

two cases: 1. when there are less than k chars in the hash table we put the char in the hash table

1. when there are exactly distinct k chars in the hash table and we need to add a new char, we delete the left most char

#

```
class Solution(object):
    def lengthOfLongestSubstringKDistinct(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: int
        """
        if k == 0:
            return 0
        start = 0
        char_hash = {}
        max_len = 0
        for char in s:
            if len(char_hash) < k:
                if char not in char_hash:
                    char_hash[char] = 1
                else:
                    char_hash[char] += 1
            elif len(char_hash) == k:
                if char not in char_hash:
                    max_len = max(max_len, sum(char_hash.values()))
                    while len(char_hash) == k:
                        char_hash[s[start]] -= 1
                        if char_hash[s[start]] == 0:
                            del char_hash[s[start]]
                        start += 1
                    char_hash[char] = 1
                else:
                    char_hash[char] += 1
        max_len = max(max_len, sum(char_hash.values()))
        return max_len
```

written by liangwang original link here

Solution 3

This problem can be solved using two pointers. The important part is `while (map.size() > k)`, we move left pointer to make sure the map size is less or equal to `k`. This can be easily extended to any number of unique characters.

```
public int lengthOfLongestSubstringKDistinct(String s, int k) {
    Map<Character, Integer> map = new HashMap<>();
    int left = 0;
    int best = 0;
    for(int i = 0; i < s.length(); i++) {
        // character at the right pointer
        char c = s.charAt(i);
        map.put(c, map.getOrDefault(c, 0) + 1);
        // make sure map size is valid, no need to check left pointer less than s
.length()
        while (map.size() > k) {
            char leftChar = s.charAt(left);
            map.put(leftChar, map.get(leftChar) - 1);
            if (map.get(leftChar) == 0) {
                map.remove(leftChar);
            }
            left++;
        }
        best = Math.max(best, i - left + 1);
    }
    return best;
}
```

written by yfcheng original link here

From Leetcoder.

Flatten Nested List Iterator

Given a nested list of integers, implement an iterator to flatten it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1:
Given the list `[[1,1],2,[1,1]]`,

By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: `[1,1,2,1,1]`.

Example 2:
Given the list `[1,[4,[6]]]`,

By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: `[1,4,6]`.

Solution 1

An iterator shouldn't copy the entire data but just iterate over the original data structure.

I keep the current progress in a stack. My `hasNext` tries to find an integer. My `next` returns it and moves on. I call `hasNext` in `next` because `hasNext` is optional. Some user of the iterator might call only `next` and never `hasNext`, e.g., if they know how many integers are in the structure or if they want to handle the ending with exception handling.

Python

Using a stack of [list, index] pairs.

```
class NestedIterator(object):

    def __init__(self, nestedList):
        self.stack = [[nestedList, 0]]

    def next(self):
        self.hasNext()
        nestedList, i = self.stack[-1]
        self.stack[-1][1] += 1
        return nestedList[i].getInteger()

    def hasNext(self):
        s = self.stack
        while s:
            nestedList, i = s[-1]
            if i == len(nestedList):
                s.pop()
            else:
                x = nestedList[i]
                if x.isInteger():
                    return True
                s[-1][1] += 1
                s.append([x.getList(), 0])
        return False
```

Java

Using a stack of ListIterators.

```java
public class NestedIterator implements Iterator<Integer> {

    public NestedIterator(List<NestedInteger> nestedList) {
        lists = new Stack<>();
        lists.push(nestedList.listIterator());
    }

    public Integer next() {
        hasNext();
        return lists.peek().next().getInteger();
    }

    public boolean hasNext() {
        while (!lists.empty()) {
            if (!lists.peek().hasNext()) {
                lists.pop();
            } else {
                NestedInteger x = lists.peek().next();
                if (x.isInteger())
                    return lists.peek().previous() == x;
                lists.push(x.getList().listIterator());
            }
        }
        return false;
    }

    private Stack<ListIterator<NestedInteger>> lists;
}
```

C++

Using stacks of begin and end iterators.

```
class NestedIterator {
public:
    NestedIterator(vector<NestedInteger> &nestedList) {
        begins.push(nestedList.begin());
        ends.push(nestedList.end());
    }

    int next() {
        hasNext();
        return (begins.top()++)->getInteger();
    }

    bool hasNext() {
        while (begins.size()) {
            if (begins.top() == ends.top()) {
                begins.pop();
                ends.pop();
            } else {
                auto x = begins.top();
                if (x->isInteger())
                    return true;
                begins.top()++;
                begins.push(x->getList().begin());
                ends.push(x->getList().end());
            }
        }
        return false;
    }

private:
    stack<vector<NestedInteger>::iterator> begins, ends;
};
```

written by StefanPochmann original link here

Solution 2

A question before this is the Nested List Weight Sum, and it requires recursion to solve. As it carries to this problem that we will need recursion to solve it. But since we need to access each NestedInteger at a time, we will use a stack to help.

In the constructor, we push all the nestedList into the stack from back to front, so when we pop the stack, it returns the very first element. Second, in the hasNext() function, we peek the first element in stack currently, and if it is an Integer, we will return true and pop the element. If it is a list, we will further flatten it. This is iterative version of flatting the nested list. Again, we need to iterate from the back to front of the list.

```
public class NestedIterator implements Iterator<Integer> {
    Stack<NestedInteger> stack = new Stack<>();
    public NestedIterator(List<NestedInteger> nestedList) {
        for(int i = nestedList.size() - 1; i >= 0; i--) {
            stack.push(nestedList.get(i));
        }
    }

    @Override
    public Integer next() {
        return stack.pop().getInteger();
    }

    @Override
    public boolean hasNext() {
        while(!stack.isEmpty()) {
            NestedInteger curr = stack.peek();
            if(curr.isInteger()) {
                return true;
            }
            stack.pop();
            for(int i = curr.getList().size() - 1; i >= 0; i--) {
                stack.push(curr.getList().get(i));
            }
        }
        return false;
    }
}
```

written by yfcheng original link here

Solution 3

For this problem "Run Code" button is not working and code can't be tested. Please fix it :)

written by kaidul original link here

From Leetcoder.

Power of Four

Given an integer (signed 32 bits), write a function to check whether it is a power of 4.

Example:
Given num = 16, return true. Given num = 5, return false.

Follow up: Could you solve it without loops/recursion?

Credits:
Special thanks to @yukuairoy for adding this problem and creating all test cases.

Solution 1

```
    public boolean isPowerOfFour(int num) {
        return num > 0 && (num&(num-1)) == 0 && (num & 0x55555555) != 0;
        //0x55555555 is to get rid of those power of 2 but not power of 4
        //so that the single 1 bit always appears at the odd position
    }
```

written by aiscong original link here

Solution 2

```
bool isPowerOfFour(int num) {
    return num > 0 && (num & (num - 1)) == 0 && (num - 1) % 3 == 0;
}
```

written by Beibeibei original link here

Solution 3

```
public class Solution {
    public boolean isPowerOfFour(int num) {
        return (num > 0) && ((num & (num - 1)) == 0) && ((num & 0x55555555) == nu
m);
    }
}
```

The basic idea is from power of 2, We can use "n&(n-1) == 0" to determine if n is power of 2. For power of 4, the additional restriction is that in binary form, the only "1" should always located at the odd position. For example, 4^0 = 1, 4^1 = 100, 4^2 = 10000. So we can use "num & 0x55555555==num" to check if "1" is located at the odd position.

written by diegozeng original link here

From Leetcoder.

Integer Break

Given a positive integer n, break it into the sum of **at least** two positive integers and maximize the product of those integers. Return the maximum product you can get.

For example, given $n = 2$, return 1 ($2 = 1 + 1$); given $n = 10$, return 36 ($10 = 3 + 3 + 4$).

Note: you may assume that n is not less than 2.

1. There is a simple O(n) solution to this problem.
2. You may check the breaking results of n ranging from 7 to 10 to discover the regularities.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 1

I saw many solutions were referring to factors of 2 and 3. But why these two magic numbers? Why other factors do not work? Let's study the math behind it.

For convenience, say **n** is sufficiently large and can be broken into any smaller real positive numbers. We now try to calculate which real number generates the largest product. Assume we break **n** into **(n / x) x**'s, then the product will be $x^{n/x}$, and we want to maximize it.

Taking its derivative gives us $n * x^{n/x-2} * (1 - \ln(x))$. The derivative is positive when $0 < x < e$, and equal to **0** when $x = e$, then becomes negative when $x > e$, which indicates that the product increases as **x** increases, then reaches its maximum when $x = e$, then starts dropping.

This reveals the fact that if **n** is sufficiently large and we are allowed to break **n** into real numbers, the best idea is to break it into nearly all **e**'s. On the other hand, if **n** is sufficiently large and we can only break **n** into integers, we should choose integers that are closer to **e**. The only potential candidates are **2** and **3** since $2 < e < 3$, but we will generally prefer **3** to **2**. Why?

Of course, one can prove it based on the formula above, but there is a more natural way shown as follows.

$6 = 2 + 2 + 2 = 3 + 3$. But $2 * 2 * 2 < 3 * 3$. Therefore, if there are three **2**'s in the decomposition, we can replace them by two **3**'s to gain a larger product.

All the analysis above assumes **n** is significantly large. When **n** is small (say $n <= 10$), it may contain flaws. For instance, when $n = 4$, we have $2 * 2 > 3 * 1$. To fix it, we keep breaking **n** into **3**'s until **n** gets smaller than **10**, then solve the problem by brute-force.

written by lixx2100 original link here

Solution 2

Given a number n lets say we have a possible product P = p1 * p2 * ...pk. *Then we notice what would happen if we could break pi up into two more terms lets say one of the terms is 2 we would get the terms pi-2 and 2 so if 2*(pi-2) > pi we would get a bigger product and this happens if pi > 4. since there is one other possible number less then 4 that is not 2 aka 3. Likewise for 3 if we instead breakup the one of the terms into pi-3 and 3 we would get a bigger product if 3*(pi-3) > pi which happens if pi > 4.5.

Hence we see that all of the terms in the product must be 2's and 3's. So we now just need to write n = a*3* + *b*2 such that P = (3^a) * (2^b) is maximized. Hence we should favor more 3's then 2's in the product then 2's if possible.

So if n = a*3 then the answer will just be 3^a.

if n = a*3* + *2 then the answer will be 2*(3^a).

and if n = a*3* + *22* then the answer will be 2 * 2 * 3^a

The above three cover all cases that n can be written as and the Math.pow() function takes O(log n) time to preform hence that is the running time.

```
public class Solution {
    public int integerBreak(int n) {
        if(n == 2)
            return 1;
        else if(n == 3)
            return 2;
        else if(n%3 == 0)
            return (int)Math.pow(3, n/3);
        else if(n%3 == 1)
            return 2 * 2 * (int) Math.pow(3, (n - 4) / 3);
        else
            return 2 * (int) Math.pow(3, n/3);
    }

}
```

written by kevin36 original link here

Solution 3

```
If we want to break a number, breaking it into 3s turns out to be the most efficie
nt.
2^3 < 3^2
4^3 < 3^4
5^3 < 3^5
6^3 < 3^6
...

Therefore, intuitively, we want as many 3 as possible
if a number % 3 == 0, we just break it into 3s -> the product is Math.pow(3, n/3)

As for numbers % 3 == 1, we don't want the 'times * 1' in the end;
    borrowing a 3 is a natural thought.
    if we borrow a 3, 3 can be divided into
         case 1: 1 + 2 -> with the extra 1, we have 2*2 = 4
         case 2: (0) + 3 -> with the extra 1, we have 4
         turns out these two cases have the same results
    so, for numbers % 3 == 1 -> the result would be Math.pow(3, n/3-1)*4

Then we have the numbers % 3 == 2 left
     again, we try to borrow a 3,
         case 1: 1+2 -> with the extra 2, we have 1*5 or 3*2 => 3*2 is better
         case 2: 0+3 -> with the extra 2, we have 2*3 or 5 => 2*3 is better
     and we actually just end up with not borrowing at all!
     so we can just *2 if we have an extra 2 -> the result would be Math.pow(3, n/
3)*2

Then, we have a couple corner cases two deal with since so far we only looked at
numbers  that are larger than 3 -> luckily, we only have 2 and 3 left,
which are pretty easy to figure out

Thus my final solution is

public class Solution {
    public int integerBreak(int n) {
        if(n <= 3) return n-1; //assuming n >= 2
        return n%3 == 0 ? (int)Math.pow(3, n/3) : n%3 == 1 ? (int)Math.pow(3, n/3
-1)*4 : (int)Math.pow(3, n/3)*2;
    }
}
```

written by aiscong original link here

From Leetcoder.

Reverse String

Write a function that takes a string as input and returns the string reversed.

Example:
Given s = "hello", return "olleh".

Solution 1

Cheating Method using StringBuilder

```
public class Solution {
    public String reverseString(String s) {
        return  new StringBuilder(s).reverse().toString();
    }
}
```

Classic Method by swapping first and last

```
public class Solution {
    public String reverseString(String s) {
        char[] word = s.toCharArray();
        int i = 0;
        int j = s.length() - 1;
        while (i < j) {
            char temp = word[i];
            word[i] = word[j];
            word[j] = temp;
            i++;
            j--;
        }
        return new String(word);
    }
}
```

Same as previous but using byte instead

```
public class Solution {
    public String reverseString(String s) {
        byte[] bytes = s.getBytes();
        int i = 0;
        int j = s.length() - 1;
        while (i < j) {
            byte temp = bytes[i];
            bytes[i] = bytes[j];
            bytes[j] = temp;
            i++;
            j--;
        }
        return new String(bytes);
    }
}
```

Classic Method by swapping first and last

If you don't like temp variable

```java
public class Solution {
    public String reverseString(String s) {
        byte[] bytes = s.getBytes();
        int i = 0;
        int j = s.length() - 1;
        while (i < j) {
            bytes[i] = (byte)(bytes[i] ^ bytes[j]);
            bytes[j] = (byte)(bytes[i] ^ bytes[j]);
            bytes[i] = (byte)(bytes[i] ^ bytes[j]);
            i++;
            j--;
        }
        return new String(bytes);
    }
}
```

Using recursion

```java
public class Solution {
    public String reverseString(String s) {
        int length = s.length();
        if (length <= 1) return s;
        String leftStr = s.substring(0, length / 2);
        String rightStr = s.substring(length / 2, length);
        return reverseString(rightStr) + reverseString(leftStr);
    }
}
```

written by ratchapong.t original link here

Solution 2

```
class Solution {
public:
    string reverseString(string s) {
        int i = 0, j = s.size() - 1;
        while(i < j){
            swap(s[i++], s[j--]);
        }

        return s;
    }
};
```

written by Ximin.Z original link here

Solution 3

```
class Solution(object):
def reverseString(self, s):
    """
    :type s: str
    :rtype: str
    """
    return s[::-1]
```

written by cheffyu original link here

From Leetcoder.

Reverse Vowels of a String

Write a function that takes a string as input and reverse only the vowels of a string.

Example 1:
Given s = "hello", return "holle".

Example 2:
Given s = "leetcode", return "leotcede".

Solution 1

In the inner while loop, don't forget the condition "start less than end" while incrementing start and decrementing end. This is my friend's google phone interview question. Cheers! // update! May use a HashSet to reduce the look up time to O(1)

```
public class Solution {
public String reverseVowels(String s) {
    if(s == null || s.length()==0) return s;
    String vowels = "aeiouAEIOU";
    char[] chars = s.toCharArray();
    int start = 0;
    int end = s.length()-1;
    while(start<end){

        while(start<end && !vowels.contains(chars[start]+"")){
            start++;
        }

        while(start<end && !vowels.contains(chars[end]+"")){
            end--;
        }

        char temp = chars[start];
        chars[start] = chars[end];
        chars[end] = temp;

        start++;
        end--;
    }
    return new String(chars);
}
```

}

written by ninacc original link here

Solution 2

Ruby

```ruby
def reverse_vowels(s)
  vowels = s.scan(/[aeiou]/i)
  s.gsub(/[aeiou]/i) { vowels.pop }
end
```

Python

```python
def reverseVowels(self, s):
    vowels = re.findall('(?i)[aeiou]', s)
    return re.sub('(?i)[aeiou]', lambda m: vowels.pop(), s)
```

It's possible in one line, but I don't really like it:

```python
def reverseVowels(self, s):
    return re.sub('(?i)[aeiou]', lambda m, v=re.findall('(?i)[aeiou]', s): v.pop(
), s)
```

Another version, finding replacement vowels on the fly instead of collecting all in advance:

```python
def reverseVowels(self, s):
    vowels = (c for c in reversed(s) if c in 'aeiouAEIOU')
    return re.sub('(?i)[aeiou]', lambda m: next(vowels), s)
```

written by StefanPochmann original link here

Solution 3

```
class Solution {
public:
    string reverseVowels(string s) {
        auto p1 = s.begin(), p2 = s.end() - 1;
        string vowels = "aeiouAEIOU";
        while(p1 < p2) {
            while((vowels.find(*p1) == string::npos) && (p1 < p2)) p1++;
            while((vowels.find(*p2) == string::npos) && (p1 < p2)) p2--;
            if(p1 < p2) swap(*p1, *p2);
            p1++;
            p2--;
        }
        return s;
    }
};
```

written by linjian2015 original link here

From Leetcoder.

Moving Average from Data Stream

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

For example,

```
MovingAverage m = new MovingAverage(3);
m.next(1) = 1
m.next(10) = (1 + 10) / 2
m.next(3) = (1 + 10 + 3) / 3
m.next(5) = (10 + 3 + 5) / 3
```

Solution 1

```
import collections

class MovingAverage(object):

    def __init__(self, size):
        """
        Initialize your data structure here.
        :type size: int
        """
        self.queue = collections.deque(maxlen=size)


    def next(self, val):
        """
        :type val: int
        :rtype: float
        """
        queue = self.queue
        queue.append(val)
        return float(sum(queue))/len(queue)



# Your MovingAverage object will be instantiated and called as such:
# obj = MovingAverage(size)
# param_1 = obj.next(val)
```

written by microleo720 original link here

Solution 2

Essentially, we just need to keep track of the sum of the current window as we go. This prevents an O(n) traversal over the Queue as we add new numbers to get the new average. If we need to evict then we just subtract that number off of our sum and divide by the size.

```
public class MovingAverage {
private double previousSum = 0.0;
private int maxSize;
private Queue<Integer> currentWindow;

public MovingAverage(int size) {
    currentWindow = new LinkedList<Integer>();
    maxSize = size;
}

public double next(int val) {
    if (currentWindow.size() == maxSize)
    {
        previousSum -= currentWindow.remove();
    }

    previousSum += val;
    currentWindow.add(val);
    return previousSum / currentWindow.size();
}}
```

written by bdwalker original link here

Solution 3

```java
public class MovingAverage {

    Deque<Integer> dq;
    int size;
    int sum;
    public MovingAverage(int size) {
        dq = new LinkedList<>();
        this.size = size;
        this.sum = 0;
    }

    public double next(int val) {
        if (dq.size() < size) {
            sum += val;
            dq.addLast(val);
            return (double) (sum / dq.size());
        } else {
            int temp = dq.pollFirst();
            sum -= temp;
            dq.addLast(val);
            sum += val;
            return (double) (sum / size);
        }
    }

}
```

written by publicstatic2 original link here

From Leetcoder.

Top K Frequent Elements

Given a non-empty array of integers, return the ***k*** most frequent elements.

For example,
Given `[1,1,1,2,2,3]` and k = 2, return `[1,2]`.

Note:

- You may assume k is always valid, $1 \le k \le$ number of unique elements.
- Your algorithm's time complexity **must be** better than $O(n \log n)$, where n is the array's size.

Solution 1

Idea is simple. Build a array of list to be buckets with length 1 to sort.

```
public List<Integer> topKFrequent(int[] nums, int k) {

    List<Integer>[] bucket = new List[nums.length + 1];
    Map<Integer, Integer> frequencyMap = new HashMap<Integer, Integer>();

    for (int n : nums) {
        frequencyMap.put(n, frequencyMap.getOrDefault(n, 0) + 1);
    }

    for (int key : frequencyMap.keySet()) {
        int frequency = frequencyMap.get(key);
        if (bucket[frequency] == null) {
            bucket[frequency] = new ArrayList<>();
        }
        bucket[frequency].add(key);
    }

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

    for (int pos = bucket.length - 1; pos >= 0 && res.size() < k; pos--) {
        if (bucket[pos] != null) {
            res.addAll(bucket[pos]);
        }
    }
    return res;
}
```

written by m010 original link here

Solution 2

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> map;
        for(int num : nums){
            map[num]++;
        }

        vector<int> res;
        // pair<first, second>: first is frequency,  second is number
        priority_queue<pair<int,int>> pq;
        for(auto it = map.begin(); it != map.end(); it++){
            pq.push(make_pair(it->second, it->first));
            if(pq.size() > (int)map.size() - k){
                res.push_back(pq.top().second);
                pq.pop();
            }
        }
        return res;
    }
};
```

written by sxycwzwzq original link here

Solution 3

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> m;
        for (int num : nums)
            ++m[num];

        vector<vector<int>> buckets(nums.size() + 1);
        for (auto p : m)
            buckets[p.second].push_back(p.first);

        vector<int> ans;
        for (int i = buckets.size() - 1; i >= 0 && ans.size() < k; --i) {
            for (int num : buckets[i]) {
                ans.push_back(num);
                if (ans.size() == k)
                    break;
            }
        }
        return ans;
    }
};
```

written by Aeonaxx original link here

From Leetcoder.

Design Tic-Tac-Toe

Design a Tic-tac-toe game that is played between two players on a n x n grid.

You may assume the following rules:

1. A move is guaranteed to be valid and is placed on an empty block.
2. Once a winning condition is reached, no more moves is allowed.
3. A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

Example:

```
Given n = 3, assume that player 1 is "X" and player 2 is "O" in the board.

TicTacToe toe = new TicTacToe(3);

toe.move(0, 0, 1); -> Returns 0 (no one wins)
|X| | |
| | | |    // Player 1 makes a move at (0, 0).
| | | |

toe.move(0, 2, 2); -> Returns 0 (no one wins)
|X| |O|
| | | |    // Player 2 makes a move at (0, 2).
| | | |

toe.move(2, 2, 1); -> Returns 0 (no one wins)
|X| |O|
| | | |    // Player 1 makes a move at (2, 2).
| | |X|

toe.move(1, 1, 2); -> Returns 0 (no one wins)
|X| |O|
| |O| |    // Player 2 makes a move at (1, 1).
| | |X|

toe.move(2, 0, 1); -> Returns 0 (no one wins)
|X| |O|
| |O| |    // Player 1 makes a move at (2, 0).
|X| |X|

toe.move(1, 0, 2); -> Returns 0 (no one wins)
|X| |O|
|O|O| |    // Player 2 makes a move at (1, 0).
|X| |X|

toe.move(2, 1, 1); -> Returns 1 (player 1 wins)
|X| |O|
|O|O| |    // Player 1 makes a move at (2, 1).
|X|X|X|
```

Follow up:
Could you do better than $O(n^2)$ per `move()` operation?

1. Could you trade extra space such that `move()` operation can be done in $O(1)$?

2. You need two arrays: int rows[n], int cols[n], plus two variables: diagonal, anti_diagonal.

Solution 1

Initially, I had not read the Hint in the question and came up with an O(n) solution. After reading the extremely helpful hint; a much easier approach became apparent. The key observation is that in order to win Tic-Tac-Toe you must have the entire row or column. Thus, we don't need to keep track of an entire n^2 board. We only need to keep a count for each row and column. If at any time a row or column matches the size of the board then that player has won.

To keep track of which player, I add one for Player1 and -1 for Player2. There are two additional variables to keep track of the count of the diagonals. Each time a player places a piece we just need to check the count of that row, column, diagonal and anti-diagonal.

Also see a very similar answer that I believe had beaten me to the punch. We came up with our solutions independently but they are very similar in principle. Aeonaxx's soln

```java
public class TicTacToe {
private int[] rows;
private int[] cols;
private int diagonal;
private int antiDiagonal;

/** Initialize your data structure here. */
public TicTacToe(int n) {
    rows = new int[n];
    cols = new int[n];
}

/** Player {player} makes a move at ({row}, {col}).
    @param row The row of the board.
    @param col The column of the board.
    @param player The player, can be either 1 or 2.
    @return The current winning condition, can be either:
            0: No one wins.
            1: Player 1 wins.
            2: Player 2 wins. */
public int move(int row, int col, int player) {
    int toAdd = player == 1 ? 1 : -1;

    rows[row] += toAdd;
    cols[col] += toAdd;
    if (row == col)
    {
        diagonal += toAdd;
    }

    if (col == (cols.length - row - 1))
    {
        antiDiagonal += toAdd;
    }

    int size = rows.length;
    if (Math.abs(rows[row]) == size ||
        Math.abs(cols[col]) == size ||
        Math.abs(diagonal) == size  ||
        Math.abs(antiDiagonal) == size)
    {
        return player;
    }

    return 0;
}
```

}

written by bdwalker original link here

Solution 2

new version:

```
class TicTacToe {
private:
    //count parameter: player 1 + : player 2: -
    vector<int> rowJudge;
    vector<int> colJudge;
    int diag, anti;
    int total;
public:
    /** Initialize your data structure here. */

    TicTacToe(int n):total(n), rowJudge(n), colJudge(n),diag(0),anti(0){}

    int move(int row, int col, int player) {
        int add = player == 1 ? 1 : -1;
        diag += row == col ? add : 0;
        anti += row == total - col - 1 ? add : 0;
        rowJudge[row] += add;
        colJudge[col] += add;
        if(abs(rowJudge[row]) == total || abs(colJudge[col]) == total || abs(diag)
== total || abs(anti) == total)
            return player;
        return 0;
    }
};
```

old version:

```
class TicTacToe {
private:
    //status:
    // 0: no one fill
    // 1 or 2: player fill
    //-1 : invalid
    //pair:
    //first:player, second:count
    vector<pair<int,int>> rowJudge;
    vector<pair<int,int>> colJudge;
    pair<int,int> diag, anti;
    int total;
public:
    /** Initialize your data structure here. */
    TicTacToe(int n):total(n), rowJudge(n), colJudge(n){}

    /** Player {player} makes a move at ({row}, {col}).
        @param row The row of the board.
        @param col The column of the board.
        @param player The player, can be either 1 or 2.
        @return The current winning condition, can be either:
                0: No one wins.
                1: Player 1 wins.
                2: Player 2 wins. */
```

```
                2: Player 2 wins. */
    int move(int row, int col, int player) {
        if(rowJudge[row].first == 0 || rowJudge[row].first == player){
            rowJudge[row].first = player;
            rowJudge[row].second++;
            if(rowJudge[row].second == total){
                return player;
            }
        }
        else {
            rowJudge[row].first = -1;
        }

        if(colJudge[col].first == 0 || colJudge[col].first == player){
            colJudge[col].first = player;
            colJudge[col].second++;
            if(colJudge[col].second == total){
                return player;
            }
        }
        else {
            colJudge[col].first = -1;
        }


        if(row == col){
            if(diag.first == 0 || diag.first == player){
                diag.first = player;
                diag.second++;
                if(diag.second == total){
                    return player;
                }
            }
            else{
                diag.first = -1;
            }
        }
        if(row + col == total - 1){
            if(anti.first == 0 || anti.first == player){
                anti.first = player;
                anti.second++;
                if(anti.second == total){
                    return player;
                }
            }
            else{
                anti.first = -1;
            }
        }
        return 0;
    }
};
```

written by sxycwzwzq original link here

Solution 3

```
public class TicTacToe {

    private int[] rows;
    private int[] cols;
    private int size;
    private int diagonal;
    private int anti_diagonal;
    /** Initialize your data structure here. */
    public TicTacToe(int n) {
        size = n;
        rows = new int[n];
        cols = new int[n];
    }

    /** Player {player} makes a move at ({row}, {col}).
        @param row The row of the board.
        @param col The column of the board.
        @param player The player, can be either 1 or 2.
        @return The current winning condition, can be either:
                0: No one wins.
                1: Player 1 wins.
                2: Player 2 wins. */
    public int move(int row, int col, int player) {
        int add = player == 1 ? 1 : -1;
        if(col == row){
            diagonal += add;
        }
        if(col == size - 1 - row){
            anti_diagonal += add;
        }
        rows[row] += add;
        cols[col] += add;
        if(Math.abs(rows[row]) == size || Math.abs(cols[col]) == size|| Math.abs(
diagonal) == size || Math.abs(anti_diagonal) == size){
            return player;
        }
        return 0;
    }
}
```

written by lei11 original link here

Intersection of Two Arrays

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

Example:
Given *nums1* = `[1, 2, 2, 1]`, *nums2* = `[2, 2]`, return `[2]`.

Note:

- Each element in the result must be unique.
- The result can be in any order.

Solution 1

Use two hash sets

Time complexity: O(n)

```
public class Solution {
    public int[] intersection(int[] nums1, int[] nums2) {
        Set<Integer> set = new HashSet<>();
        Set<Integer> intersect = new HashSet<>();
        for (int i = 0; i < nums1.length; i++) {
            set.add(nums1[i]);
        }
        for (int i = 0; i < nums2.length; i++) {
            if (set.contains(nums2[i])) {
                intersect.add(nums2[i]);
            }
        }
        int[] result = new int[intersect.size()];
        int i = 0;
        for (Integer num : intersect) {
            result[i++] = num;
        }
        return result;
    }
}
```

Sort both arrays, use two pointers

Time complexity: O(nlogn)

```java
public class Solution {
    public int[] intersection(int[] nums1, int[] nums2) {
        Set<Integer> set = new HashSet<>();
        Arrays.sort(nums1);
        Arrays.sort(nums2);
        int i = 0;
        int j = 0;
        while (i < nums1.length && j < nums2.length) {
            if (nums1[i] < nums2[j]) {
                i++;
            } else if (nums1[i] > nums2[j]) {
                j++;
            } else {
                set.add(nums1[i]);
                i++;
                j++;
            }
        }
        int[] result = new int[set.size()];
        int k = 0;
        for (Integer num : set) {
            result[k++] = num;
        }
        return result;
    }
}
```

Binary search

Time complexity: O(nlogn)

```java
public class Solution {
    public int[] intersection(int[] nums1, int[] nums2) {
        Set<Integer> set = new HashSet<>();
        Arrays.sort(nums2);
        for (Integer num : nums1) {
            if (binarySearch(nums2, num)) {
                set.add(num);
            }
        }
        int i = 0;
        int[] result = new int[set.size()];
        for (Integer num : set) {
            result[i++] = num;
        }
        return result;
    }

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

written by divingboy89 original link here

Solution 2

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> m(nums1.begin(), nums1.end());
        vector<int> res;
        for (auto a : nums2)
            if (m.count(a)) {
                res.push_back(a);
                m.erase(a);
            }
        return res;
    }
};
```

written by Kenigma original link here

Solution 3

```
vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
    set<int> s(nums1.begin(), nums1.end());
    vector<int> out;
    for (int x : nums2)
        if (s.erase(x))
            out.push_back(x);
    return out;
}
```

written by StefanPochmann original link here

From Leetcoder.

Intersection of Two Arrays II

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

Example:
Given *nums1* = `[1, 2, 2, 1]`, *nums2* = `[2, 2]`, return `[2, 2]`.

Note:

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

Follow up:

- What if the given array is already sorted? How would you optimize your algorithm?
- What if *nums1*'s size is small compared to *num2*'s size? Which algorithm is better?
- What if elements of *nums2* are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

Solution 1

m: nums1.size n: nums2.size

Hash table solution: Time: O(m + n) Space: O(m + n)

```
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        unordered_map<int, int> dict;
        vector<int> res;
        for(int i = 0; i < (int)nums1.size(); i++) dict[nums1[i]]++;
        for(int i = 0; i < (int)nums2.size(); i++)
            if(--dict[nums2[i]] >= 0) res.push_back(nums2[i]);
        return res;
    }
};
```

Hash table solution2: Time: O(m + n) Space: O(m)

```
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        unordered_map<int, int> dict;
        vector<int> res;
        for(int i = 0; i < (int)nums1.size(); i++) dict[nums1[i]]++;
        for(int i = 0; i < (int)nums2.size(); i++)
            if(dict.find(nums2[i]) != dict.end() && --dict[nums2[i]] >= 0) res.pu
sh_back(nums2[i]);
        return res;
    }
};
```

Sort and two pointers Solution: Time: O(max(m, n) log(max(m, n))) Space: O(m + n)

```
class Solution {
public:
    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
        sort(nums1.begin(), nums1.end());
        sort(nums2.begin(), nums2.end());
        int n1 = (int)nums1.size(), n2 = (int)nums2.size();
        int i1 = 0, i2 = 0;
        vector<int> res;
        while(i1 < n1 && i2 < n2){
            if(nums1[i1] == nums2[i2]) {
                res.push_back(nums1[i1]);
                i1++;
                i2++;
            }
            else if(nums1[i1] > nums2[i2]){
                i2++;
            }
            else{
                i1++;
            }
        }
        return res;
    }
};
```

written by sxycwzwzq original link here

Solution 2

```
public class Solution {
    public int[] intersect(int[] nums1, int[] nums2) {
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        ArrayList<Integer> result = new ArrayList<Integer>();
        for(int i = 0; i < nums1.length; i++)
        {
            if(map.containsKey(nums1[i])) map.put(nums1[i], map.get(nums1[i])+1);
            else map.put(nums1[i], 1);
        }

        for(int i = 0; i < nums2.length; i++)
        {
            if(map.containsKey(nums2[i]) && map.get(nums2[i]) > 0)
            {
                result.add(nums2[i]);
                map.put(nums2[i], map.get(nums2[i])-1);
            }
        }

       int[] r = new int[result.size()];
       for(int i = 0; i < result.size(); i++)
       {
           r[i] = result.get(i);
       }

       return r;
    }
}
```

written by VanillaCoke original link here

Solution 3

> What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

- If only nums2 cannot fit in memory, put all elements of nums1 into a HashMap, read chunks of array that fit into the memory, and record the intersections.
- If both nums1 and nums2 are so huge that neither fit into the memory, sort them individually (external sort), then read 2 elements from each array at a time in memory, record intersections.

written by vishalhemnani original link here

From Leetcoder.

Android Unlock Patterns

Given an Android **3x3** key lock screen and two integers **m** and **n**, where 1 ≤ m ≤ n ≤ 9, count the total number of unlock patterns of the Android lock screen, which consist of minimum of **m** keys and maximum **n** keys.

Rules for a valid pattern:

1. Each pattern must connect at least **m** keys and at most **n** keys.
2. All the keys must be distinct.
3. If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.
4. The order of keys used matters.

Explanation:

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

Invalid move: `4 – 1 – 3 – 6`
Line 1 - 3 passes through key 2 which had not been selected in the pattern.

Invalid move: `4 – 1 – 9 – 2`
Line 1 - 9 passes through key 5 which had not been selected in the pattern.

Valid move: `2 – 4 – 1 – 3 – 6`
Line 1 - 3 is valid because it passes through key 2, which had been selected in the pattern

Valid move: `6 – 5 – 4 – 1 – 9 – 2`
Line 1 - 9 is valid because it passes through key 5, which had been selected in the pattern.

Example:
Given **m** = 1, **n** = 1, return 9.

Credits:
Special thanks to @elmirap for adding this problem and creating all test cases.

Solution 1

The optimization idea is that 1,3,7,9 are symmetric, 2,4,6,8 are also symmetric. Hence we only calculate one among each group and multiply by 4.

```
public class Solution {
    // cur: the current position
    // remain: the steps remaining
    int DFS(boolean vis[], int[][] skip, int cur, int remain) {
        if(remain < 0) return 0;
        if(remain == 0) return 1;
        vis[cur] = true;
        int rst = 0;
        for(int i = 1; i <= 9; ++i) {
            // If vis[i] is not visited and (two numbers are adjacent or skip num
ber is already visited)
            if(!vis[i] && (skip[i][cur] == 0 || (vis[skip[i][cur]]))) {
                rst += DFS(vis, skip, i, remain - 1);
            }
        }
        vis[cur] = false;
        return rst;
    }

    public int numberOfPatterns(int m, int n) {
        // Skip array represents number to skip between two pairs
        int skip[][] = new int[10][10];
        skip[1][3] = skip[3][1] = 2;
        skip[1][7] = skip[7][1] = 4;
        skip[3][9] = skip[9][3] = 6;
        skip[7][9] = skip[9][7] = 8;
        skip[1][9] = skip[9][1] = skip[2][8] = skip[8][2] = skip[3][7] = skip[7]
[3] = skip[4][6] = skip[6][4] = 5;
        boolean vis[] = new boolean[10];
        int rst = 0;
        // DFS search each length from m to n
        for(int i = m; i <= n; ++i) {
            rst += DFS(vis, skip, 1, i - 1) * 4;    // 1, 3, 7, 9 are symmetric
            rst += DFS(vis, skip, 2, i - 1) * 4;    // 2, 4, 6, 8 are symmetric
            rst += DFS(vis, skip, 5, i - 1);        // 5
        }
        return rst;
    }
}
```

written by zzz1322 original link here

Solution 2

The general idea is DFS all the possible combinations from 1 to 9 and skip invalid moves along the way.

We can check invalid moves by using a jumping table. e.g. If a move requires a jump and the key that it is crossing is not visited, then the move is invalid. Furthermore, we can utilize symmetry to reduce runtime, in this case it reduced from ~120ms to ~70ms.

I felt clueless when first encountered this problem, and considered there must be lots of edge cases. Turns out, it's pretty straight forward. Hope this solution helps :D

```
private int[][] jumps;
private boolean[] visited;

public int numberOfPatterns(int m, int n) {
    jumps = new int[10][10];
    jumps[1][3] = jumps[3][1] = 2;
    jumps[4][6] = jumps[6][4] = 5;
    jumps[7][9] = jumps[9][7] = 8;
    jumps[1][7] = jumps[7][1] = 4;
    jumps[2][8] = jumps[8][2] = 5;
    jumps[3][9] = jumps[9][3] = 6;
    jumps[1][9] = jumps[9][1] = jumps[3][7] = jumps[7][3] = 5;
    visited = new boolean[10];
    int count = 0;
    count += DFS(1, 1, 0, m, n) * 4; // 1, 3, 7, 9 are symmetrical
    count += DFS(2, 1, 0, m, n) * 4; // 2, 4, 6, 8 are symmetrical
    count += DFS(5, 1, 0, m, n);
    return count;
}

private int DFS(int num, int len, int count, int m, int n) {
    if (len >= m) count++;
    len++;
    if (len > n) return count;
    visited[num] = true;
    for (int next = 1; next <= 9; next++) {
        int jump = jumps[num][next];
        if (!visited[next] && (jump == 0 || visited[jump])) {
            count = DFS(next, len, count, m, n);
        }
    }
    visited[num] = false; // backtracking
    return count;
}
```

written by david.liu original link here

Solution 3

If we use the symmetry, we can only start from 1, 2 and 5 then multiply the results of 1 and 2 by 4. (170ms) Thanks to @woaizuguo999

```
int res=0;
public int numberOfPatterns(int m, int n) {
    boolean[][] keyboard = new boolean[3][3];
    int ret=0;
    for (int p=m;p<=n;p++){
        for (int i=0;i<2;i++){
            for (int j=0;j<2;j++){
                if (j == 0 && i == 1) continue;
                keyboard[i][j] = true;
                helper(keyboard,p-1,i,j);
                keyboard[i][j] = false;
                ret += (i == 1 && j == 1)? res:4*res;
                res=0;
            }
        }
    }
    return ret;
}
private void helper(boolean[][] keyboard,int left, int x, int y){
    if (left == 0){
        res++;
        return;
    }
    for (int i=0;i<3;i++){
        for (int j=0;j<3;j++){
            if (keyboard[i][j]
                ||  (x==i && Math.abs(y-j)>1) && !keyboard[x][1]
                ||  (y==j && Math.abs(x-i)>1) && !keyboard[1][y]
                ||  (x+y == i+j) && Math.abs(x-i) >1 && !keyboard[1][1]
                ||  (x-y == i-j) && Math.abs(x-i) >1 && !keyboard[1][1]
                ||  (x == i && y == j)) {
                continue;
            }
            else{
                keyboard[i][j] = true;
                helper(keyboard,left-1,i,j);
                keyboard[i][j] = false;
            }
        }
    }
}
```

And we can continue improving the performance by using symmetry in step 1, which is the next step after start. For start from 1, only consider 2 6 and 5. For start from 2, only consider 3,6,9 and 5. For start from 5, only consider 1 and 2.(72 ms)

```
int res=0;
public int numberOfPatterns(int m, int n) {
    boolean[][] keyboard = new boolean[3][3];
    int ret=0;
```

```
    for (int p=m;p<=n;p++){
        for (int i=0;i<2;i++){
            for (int j=0;j<2;j++){
                if (j == 0 && i == 1) continue;
                keyboard[i][j] = true;
                helper(keyboard,p-1,i,j,true);
                keyboard[i][j] = false;
                ret += (i == 1 && j == 1)? res:4*res;
                res=0;
            }
        }
    }
    return ret;
}

private void dfshelper(boolean[][] keyboard, int left, int x, int y){
    keyboard[x][y] = true;
    helper(keyboard,left,x,y,false);
    keyboard[x][y] = false;
}

private void helper(boolean[][] keyboard,int left, int x, int y, boolean step1){
    if (left == 0){
        res++;
        return;
    }
    if (step1){
        if (x == 0 && y == 0){
            dfshelper(keyboard,left-1,0,1);
            dfshelper(keyboard,left-1,1,2);
            int temp = 2*res;
            res = 0;
            dfshelper(keyboard,left-1,1,1);
            res += temp;
        }
        if (x == 0 && y == 1){
            dfshelper(keyboard,left-1,0,2);
            dfshelper(keyboard,left-1,1,2);
            dfshelper(keyboard,left-1,2,2);
            int temp = 2*res;
            res = 0;
            dfshelper(keyboard,left-1,1,1);
            res += temp;
        }
        if (x == 1 && y == 1){
            dfshelper(keyboard,left-1,0,0);
            dfshelper(keyboard,left-1,0,1);
            res=res*4;
        }
    }
    else{
        for (int i=0;i<3;i++){
            for (int j=0;j<3;j++){
                if (keyboard[i][j]
                    ||  (x==i && Math.abs(y-j)>1) && !keyboard[x][1]
                    ||  (y==j && Math.abs(x-i)>1) && !keyboard[1][y]
                    ||  (x+y == i+j) && Math.abs(x-i) >1 && !keyboard[1][1]
```

```
                    ||  (x+y == i+j) && Math.abs(x-i) >1 && !keyboard[1][1]
                    ||  (x-y == i-j) && Math.abs(x-i) >1 && !keyboard[1][1]
                    ||  (x == i && y == j)) {
                    continue;
                }
                else{
                    keyboard[i][j] = true;
                    helper(keyboard,left-1,i,j,false);
                    keyboard[i][j] = false;
                }
            }
        }
    }
}
```

written by qwe5103368 original link here

From Leetcoder.

Data Stream as Disjoint Intervals

Given a data stream input of non-negative integers $a_1, a_2, \ldots, a_n, \ldots$, summarize the numbers seen so far as a list of disjoint intervals.

For example, suppose the integers from the data stream are 1, 3, 7, 2, 6, ..., then the summary will be:

```
[1, 1]
[1, 1], [3, 3]
[1, 1], [3, 3], [7, 7]
[1, 3], [7, 7]
[1, 3], [6, 7]
```

Follow up:
What if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size?

Credits:
Special thanks to @yunhong for adding this problem and creating most of the test cases.

Solution 1

Use TreeMap to easily find the lower and higher keys, the key is the start of the interval. Merge the lower and higher intervals when necessary. The time complexity for adding is O(logN) since lowerKey(), higherKey(), put() and remove() are all O(logN). It would be O(N) if you use an ArrayList and remove an interval from it.

```
public class SummaryRanges {
    TreeMap<Integer, Interval> tree;

    public SummaryRanges() {
        tree = new TreeMap<>();
    }

    public void addNum(int val) {
        if(tree.containsKey(val)) return;
        Integer l = tree.lowerKey(val);
        Integer h = tree.higherKey(val);
        if(l != null && h != null && tree.get(l).end + 1 == val && h == val + 1)
{
            tree.get(l).end = tree.get(h).end;
            tree.remove(h);
        } else if(l != null && tree.get(l).end + 1 >= val) {
            tree.get(l).end = Math.max(tree.get(l).end, val);
        } else if(h != null && h == val + 1) {
            tree.put(val, new Interval(val, tree.get(h).end));
            tree.remove(h);
        } else {
            tree.put(val, new Interval(val, val));
        }
    }

    public List<Interval> getIntervals() {
        return new ArrayList<>(tree.values());
    }
}
```

written by qianzhige original link here

Solution 2

In general case, vector is OK, it will take O(n) time in each add, and O(1) in get result. But if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size, we'd better use another data structure "set", because the insert operation in vector will cost O(n) time, but it only cost O(log n) in binary search tree, but it will cost O(n) time in getInterval. So use which data structure will depends.

first one is the solution use vector

```
class SummaryRanges {
public:
    void addNum(int val) {
        auto Cmp = [](Interval a, Interval b) { return a.start < b.start; };
        auto it = lower_bound(vec.begin(), vec.end(), Interval(val, val), Cmp);
        int start = val, end = val;
        if(it != vec.begin() && (it-1)->end+1 >= val) it--;
        while(it != vec.end() && val+1 >= it->start && val-1 <= it->end)
        {
            start = min(start, it->start);
            end = max(end, it->end);
            it = vec.erase(it);
        }
        vec.insert(it,Interval(start, end));
    }

    vector<Interval> getIntervals() {
        return vec;
    }
private:
    vector<Interval> vec;
};
```

and below is another solution use binary search tree.

```
class SummaryRanges {
public:
    /** Initialize your data structure here. */
    void addNum(int val) {
        auto it = st.lower_bound(Interval(val, val));
        int start = val, end = val;
        if(it != st.begin() && (--it)->end+1 < val) it++;
        while(it != st.end() && val+1 >= it->start && val-1 <= it->end)
        {
            start = min(start, it->start);
            end = max(end, it->end);
            it = st.erase(it);
        }
        st.insert(it,Interval(start, end));
    }

    vector<Interval> getIntervals() {
        vector<Interval> result;
        for(auto val: st) result.push_back(val);
        return result;
    }
private:
    struct Cmp{
        bool operator()(Interval a, Interval b){ return a.start < b.start; }
    };
    set<Interval, Cmp> st;
};
```

written by goodliar original link here

Solution 3

For a new number n, find and return the index of interval [s, t] such that s is the largest 'start' that is smaller than n. If no such interval exists, return -1. This is done using binary search.

For example,

- new number 5, intervals [[1,1], [4,6], [8,8]], binary search returns 1.
- new number 0, intervals [[1,1], [4,6], [8,8]], binary search returns -1.

After we find this 'index', there are three circumstances:

1. intervals[index] already contains val. Do nothing.
2. val can be merged into intervals[index+1]. Modify intervals[index+1].start to val.
3. val can be merged into intervals[index]. Modify intervals[index].end to val.
4. val can't be merged into either interval. Insert Interval(val, val).

Finally, after inserting val, we need to check whether intervals[index] and intervals[index+1] can be merged.

```
class SummaryRanges {
private:
    vector<Interval> intervals = vector<Interval>();

    int binarySearch(vector<Interval> intervals, int val) {
        return binarySearchHelper(intervals, 0, intervals.size(), val);
    }

    int binarySearchHelper(vector<Interval> intervals, int start, int end, int va
l) {
        if (start == end) return -1;
        if (start+1 == end && intervals[start].start < val) return start;

        int mid = (start + end)/2;
        if (intervals[mid].start == val) {
            return mid;
        } else if (intervals[mid].start < val) {
            return binarySearchHelper(intervals, mid, end, val);
        } else { //intervals[mid] > val
            return binarySearchHelper(intervals, start, mid, val);
        }
    }

public:
    /** Initialize your data structure here. */
    SummaryRanges() {

    }

    /** For a new number n, find the last(biggest) interval
     *  [s,t], such that s < n. If no such interval exists,
     *  return -1
```

```
     * return -1.
     */
    void addNum(int val) {
        int index = binarySearch(intervals, val);

        // intervals[index] contains val
        if (index != -1 && intervals[index].end >= val) {
            return;
        }

        if (index != intervals.size()-1 && val + 1 == intervals[index+1].start) {
            intervals[index+1].start = val;
        } else if (index != -1 && val - 1 == intervals[index].end) {
            intervals[index].end = val;
        } else {
            intervals.insert(intervals.begin() + index + 1, Interval(val, val));
        }

        //merge intervals[index] with intervals[index+1]
        if (index != -1 && intervals[index].end + 1 == intervals[index+1].start)
{
            intervals[index].end = intervals[index+1].end;
            intervals.erase(intervals.begin()+index+1);
        }

        return;
    }

    vector<Interval> getIntervals() {
        return this->intervals;
    }
};
```

written by stellalai original link here

From Leetcoder.

Design Snake Game

Design a Snake game that is played on a device with screen size =*width* x *height*. Play the game online if you are not familiar with the game.

The snake is initially positioned at the top left corner (0,0) with length = 1 unit.

You are given a list of food's positions in row-column order. When a snake eats the food, its length and the game's score both increase by 1.

Each food appears one by one on the screen. For example, the second food will not appear until the first food was eaten by the snake.

When a food does appear on the screen, it is guaranteed that it will not appear on a block occupied by the snake.

Example:

```
Given width = 3, height = 2, and food = [[1,2],[0,1]].

Snake snake = new Snake(width, height, food);

Initially the snake appears at position (0,0) and the food at (1,2).

|S| | |
| | |F|

snake.move("R"); -> Returns 0

| |S| |
| | |F|

snake.move("D"); -> Returns 0

| | | |
| |S|F|

snake.move("R"); -> Returns 1 (Snake eats the first food and right after that, the
second food appears at (0,1) )

| |F| |
| |S|S|

snake.move("U"); -> Returns 1

| |F|S|
| | |S|

snake.move("L"); -> Returns 2 (Snake eats the second food)

| |S|S|
| | |S|

snake.move("U"); -> Returns -1 (Game over because snake collides with border)
```

Credits:
Special thanks to @elmirap for adding this problem and creating all test cases.

Solution 1

```
class SnakeGame {
public:
/** Initialize your data structure here.
@param width - screen width
@param height - screen height
@param food - A list of food positions
E.g food = [[1,1], [1,0]] means the first food is positioned at [1,1], the second
is at [1,0]. */

int w, h, pos;
vector<pair<int, int>> food;
set<pair<int, int>> hist;
deque<pair<int, int>> q;

SnakeGame(int width, int height, vector<pair<int, int>> food) {
    this->food = food;
    w = width, h = height, pos = 0;
    q.push_back(make_pair(0, 0));
}

/** Moves the snake.
@param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D' = Down
@return The game's score after the move. Return -1 if game over.
Game over when snake crosses the screen boundary or bites its body. */
int move(string direction) {
    int row = q.back().first, col = q.back().second;
    pair<int, int> d = q.front(); q.pop_front();
    hist.erase(d);

    if (direction == "U")
        row--;
    else if (direction == "D")
        row++;
    else if (direction == "L")
        col--;
    else if (direction == "R")
        col++;

    if (row < 0 || col < 0 || col >= w || row >= h || hist.count(make_pair(row, co
l)))
        return -1;

    hist.insert(make_pair(row, col));
    q.push_back(make_pair(row, col));

    if (pos >= food.size())
        return q.size() - 1;

    if (row == food[pos].first && col == food[pos].second) {
        pos++;
        q.push_front(d);
        hist.insert(d);
    }
```

```
    return q.size() - 1;
  }
};
```

written by jaewoo original link here

Solution 2

public class SnakeGame {

```
    class Position{
        int x;
        int y;
        public Position(int x,int y){
            this.x = x;
            this.y = y;
        }
        public boolean isEqual(Position p){
            return this.x==p.x && this.y == p.y ;
        }
    }
    int len;
    int rows ,cols;

    int[][] food;
    LinkedList<Position> snake;

    /** Initialize your data structure here.
        @param width - screen width
        @param height - screen height
        @param food - A list of food positions
        E.g food = [[1,1], [1,0]] means the first food is positioned at [1,1], th
e second is at [1,0]. */
    public SnakeGame(int width, int height, int[][] food) {
        this.rows = height;
        this.cols = width;
        this.food = food;

        snake = new LinkedList<Position>();
        snake.add(new Position(0,0));
        len = 0;
    }

    /** Moves the snake.
        @param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D' = Down
        @return The game's score after the move. Return -1 if game over.
        Game over when snake crosses the screen boundary or bites its body. */
    public int move(String direction) {
        //if(len>=food.length) return len;

        Position cur = new Position(snake.get(0).x,snake.get(0).y);

        switch(direction){
        case "U":
            cur.x--;  break;
        case "L":
            cur.y--; break;
        case "R":
            cur.y++;   break;
        case "D":
            cur.x++;   break;
        }
```

```
        if(cur.x<0 || cur.x>= rows || cur.y<0 || cur.y>=cols) return -1;


        for(int i=1;i<snake.size()-1;i++){
            Position next = snake.get(i);
            if(next.isEqual(cur)) return -1;
        }
        snake.addFirst(cur);
        if(len<food.length){
            Position p = new Position(food[len][0],food[len][1]);
            if(cur.isEqual(p)){
                len++;
            }
        }
        while(snake.size()>len+1) snake.removeLast();

        return len;
    }


/**
 * Your SnakeGame object will be instantiated and called as such:
 * SnakeGame obj = new SnakeGame(width, height, food);
 * int param_1 = obj.move(direction);
 */
```

}

written by juanren original link here

Solution 3

Why the output is : [null,0,1,1,1,1,2,2,2,2,3,4,4,4,4,-1] instead of [null,0,1,1,1,1,2,2,2,2,3,4,4,4,4,4] for case: ["SnakeGame","move","move","move","move","move","move","move","move","move","move","move","move","move","move"] [[3,3,[[2,0],[0,0],[0,2],[0,1],[2,2],[0,1]]],["D"],["D"],["R"],["U"],["U"],["L"],["D"],["R"],["R"],["U"],["L"],["L"],["D"],["R"],["U"]]

I think the last return value of move should still be 4, why it is out the boundary??

written by Yun.Hu original link here

From Leetcoder.

Russian Doll Envelopes

You have a number of envelopes with widths and heights given as a pair of integers `(w, h)`. One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

Example:
Given envelopes = `[[5,4],[6,4],[6,7],[2,3]]`, the maximum number of envelopes you can Russian doll is `3` ([2,3] => [5,4] => [6,7]).

Solution 1

1. Sort the array. Ascend on width and descend on height if width are same.
2. Find the longest increasing subsequence based on height.

- Since the width is increasing, we only need to consider height.
- [3, 4] cannot contains [3, 3], so we need to put [3, 4] before [3, 3] when sorting otherwise it will be counted as an increasing number if the order is [3, 3], [3, 4]

```
public int maxEnvelopes(int[][] envelopes) {
    if(envelopes == null || envelopes.length == 0
       || envelopes[0] == null || envelopes[0].length != 2)
        return 0;
    Arrays.sort(envelopes, new Comparator<int[]>(){
        public int compare(int[] arr1, int[] arr2){
            if(arr1[0] == arr2[0])
                return arr2[1] - arr1[1];
            else
                return arr1[0] - arr2[0];
       }
    });
    int dp[] = new int[envelopes.length];
    int len = 0;
    for(int[] envelope : envelopes){
        int index = Arrays.binarySearch(dp, 0, len, envelope[1]);
        if(index < 0)
            index = -(index + 1);
        dp[index] = envelope[1];
        if(index == len)
            len++;
    }
    return len;
}
```

written by TianhaoSong original link here

Solution 2

```
public int maxEnvelopes(int[][] envelopes) {
    if (   envelopes           == null
        || envelopes.length    == 0
        || envelopes[0]        == null
        || envelopes[0].length == 0){
        return 0;
    }

    Arrays.sort(envelopes, new Comparator<int[]>(){
        @Override
        public int compare(int[] e1, int[] e2){
            return Integer.compare(e1[0], e2[0]);
        }
    });

    int   n  = envelopes.length;
    int[] dp = new int[n];

    int ret = 0;
    for (int i = 0; i < n; i++){
        dp[i] = 1;

        for (int j = 0; j < i; j++){
            if (   envelopes[i][0] > envelopes[j][0]
                && envelopes[i][1] > envelopes[j][1]){
                dp[i] = Math.max(dp[i], 1 + dp[j]);
            }
        }

        ret = Math.max(ret, dp[i]);
    }
    return ret;
}
```

written by larrywang2014 original link here

Solution 3

See more explanation in Longest Increasing Subsequence Size (N log N)

```
def maxEnvelopes(self, envelopes):
    def bin_search(A, key):
        l, r = 0, len(A)
        while l < r:
            mid = (l+r)/2
            if A[mid][1] < key[1]:
                l = mid + 1
            else:
                r = mid
        return l
    envelopes.sort(
        cmp = lambda x,y: x[0]-y[0] if x[0] != y[0] else y[1]-x[1])
    n = len(envelopes)
    tails = []
    for i in range(n):
        e = envelopes[i]
        p = bin_search(tails, e)
        if p == len(tails):
            tails.append(e)
        else:
            tails[p] = e
    return len(tails)
```

written by JWZH original link here

From Leetcoder.

Design Twitter

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user and is able to see the 10 most recent tweets in the user's news feed. Your design should support the following methods:

1. **postTweet(userId, tweetId)**: Compose a new tweet.
2. **getNewsFeed(userId)**: Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent.
3. **follow(followerId, followeeId)**: Follower follows a followee.
4. **unfollow(followerId, followeeId)**: Follower unfollows a followee.

Example:

```
Twitter twitter = new Twitter();

// User 1 posts a new tweet (id = 5).
twitter.postTweet(1, 5);

// User 1's news feed should return a list with 1 tweet id -> [5].
twitter.getNewsFeed(1);

// User 1 follows user 2.
twitter.follow(1, 2);

// User 2 posts a new tweet (id = 6).
twitter.postTweet(2, 6);

// User 1's news feed should return a list with 2 tweet ids -> [6, 5].
// Tweet id 6 should precede tweet id 5 because it is posted after tweet id 5.
twitter.getNewsFeed(1);

// User 1 unfollows user 2.
twitter.unfollow(1, 2);

// User 1's news feed should return a list with 1 tweet id -> [5],
// since user 1 is no longer following user 2.
twitter.getNewsFeed(1);
```

Solution 1

```
class Twitter(object):

    def __init__(self):
        self.timer = itertools.count(step=-1)
        self.tweets = collections.defaultdict(collections.deque)
        self.followees = collections.defaultdict(set)

    def postTweet(self, userId, tweetId):
        self.tweets[userId].appendleft((next(self.timer), tweetId))

    def getNewsFeed(self, userId):
        tweets = heapq.merge(*(self.tweets[u] for u in self.followees[userId] | {
userId}))
        return [t for _, t in itertools.islice(tweets, 10)]

    def follow(self, followerId, followeeId):
        self.followees[followerId].add(followeeId)

    def unfollow(self, followerId, followeeId):
        self.followees[followerId].discard(followeeId)
```

written by StefanPochmann original link here

Solution 2

```
private static class Tweet {
    int timestamp;
    int tweetId;

    public Tweet(int tweetId, int timestamp) {
        this.tweetId = tweetId;
        this.timestamp = timestamp;
    }
}

private Map<Integer, Set<Integer>> followMap = new HashMap<Integer, Set<Integer>>
();
private Map<Integer, List<Tweet>> tweetMap = new HashMap<Integer, List<Tweet>>();

private AtomicInteger timestamp;

/** Initialize your data structure here. */
public Twitter() {
    timestamp = new AtomicInteger(0);
}

/** Compose a new tweet. */
public void postTweet(int userId, int tweetId) {
    Tweet newTweet = new Tweet(tweetId, timestamp.getAndIncrement());

    if (!tweetMap.containsKey(userId)) {
        tweetMap.put(userId, new ArrayList<Tweet>());  //Assuming no deletion for
now?
    }

    tweetMap.get(userId).add(newTweet);
}

/**
 * Retrieve the 10 most recent tweet ids in the user's news feed. Each item
 * in the news feed must be posted by users who the user followed or by the
 * user herself. Tweets must be ordered from most recent to least recent.
 */
public List<Integer> getNewsFeed(int userId) {
    List<Integer> result = new ArrayList<Integer>(10);

    PriorityQueue<int[]> pq = new PriorityQueue<int[]>(new Comparator<int[]>() {
        public int compare(int[] it1, int[] it2) {
            return tweetMap.get(it2[0]).get(it2[1]).timestamp - tweetMap.get(it1[
0]).get(it1[1]).timestamp;
        }
    });

    Set<Integer> followeeSet = new HashSet<Integer>();
    followeeSet.add(userId);
    if (followMap.containsKey(userId)) {
        followeeSet.addAll(followMap.get(userId));
    }
```

```java
    for (Integer followee : followeeSet) {
        if (tweetMap.containsKey(followee)) {
            List<Tweet> tweetList = tweetMap.get(followee);
            if (tweetList.size() > 0) {
                pq.add(new int[] { followee, tweetList.size() - 1 });
            }
        }
    }

    while (result.size() < 10 && pq.size() > 0) {
        int[] it = pq.poll();

        result.add(tweetMap.get(it[0]).get(it[1]).tweetId);
        it[1]--;
        if (it[1] >= 0) {
            pq.add(it);
        }
    }

    return result;
}

/**
 * Follower follows a followee. If the operation is invalid, it should be a
 * no-op.
 */
public void follow(int followerId, int followeeId) {
    Set<Integer> followSet = followMap.get(followerId);
    if (followSet == null) {
        followSet = new HashSet<Integer>();
        followMap.put(followerId, followSet);
    }

    followSet.add(followeeId);
}

/**
 * Follower unfollows a followee. If the operation is invalid, it should be
 * a no-op.
 */
public void unfollow(int followerId, int followeeId) {
    Set<Integer> followSet = followMap.get(followerId);
    if (followSet == null) {
        followSet = new HashSet<Integer>();
        followMap.put(followerId, followSet);
    }

    followSet.remove(followeeId);
}
```

written by lop original link here

Line Reflection

Given n points on a 2D plane, find if there is such a line parallel to y-axis that reflect the given set of points.

Example 1:

Given *points* = `[[1,1],[-1,1]]`, return `true`.

Example 2:

Given *points* = `[[1,1],[-1,-1]]`, return `false`.

Credits:
Special thanks to @memoryless for adding this problem and creating all test cases.

Solution 1

Idea: Reflect the points by replacing every x with minX+maxX-x and then check whether you get the same points. Why minX+maxX-x? I actually thought of it as minX+(maxX-x), i.e., first the subtraction (maxX-x). That's how far x is away from the max, so instead go that distance from the min.

Ruby

```
def is_reflected(points)
  points.sort! == points.map { |x, y| [points[0][0] + points[-1][0] - x, y] }.sor
t
end
```

Python

```
def isReflected(self, points):
    points.sort()
    return points == sorted([points[0][0] + points[-1][0] - x, y]
                            for x, y in points)
```

A linear time one:

```
def isReflected(self, points):
    if not points: return True
    X = min(points)[0] + max(points)[0]
    return {(x, y) for x, y in points} == {(X - x, y) for x, y in points}
```

Shorter, but I think less nice:

```
    return set(map(tuple, points)) == {(X - x, y) for x, y in points}
```

written by StefanPochmann original link here

Solution 2

The idea is quite simple. If there exists a line reflecting the points, then each pair of symmetric points will have their x coordinates adding up to the same value, including the pair with the maximum and minimum x coordinates. So, in the first pass, I iterate through the array, adding each point to the hash set, and keeping record of the minimum and maximum x coordinates. Then, in the second pass, I check for every point to the left of the reflecting line, if its symmetric point is in the point set or not. If all points pass the test, then there exists a reflecting line. Otherwise, not.

By the way, here, to hash the content of an array, rather than the reference value, I use **Arrays.hashCode(int[])** first, and then re-hash this hash code. You can also use **Arrays.toString(int[])** to first convey the 2d array to a string, and then hash the string. But the second method is slower.

```
public class Solution {
    public boolean isReflected(int[][] points) {
        HashSet<Integer> pointSet = new HashSet<>();
        int sum;
        int maxX, minX;

        minX = Integer.MAX_VALUE;
        maxX = Integer.MIN_VALUE;
        for(int[] point:points) {
            maxX = Math.max(maxX, point[ 0 ]);
            minX = Math.min(minX, point[ 0 ]);
            pointSet.add(Arrays.hashCode(point));
        }

        sum = maxX+minX;
        for(int[] point:points) {
            if(!pointSet.contains(Arrays.hashCode(new int[]{sum-point[ 0 ], point
[ 1 ]}))) {
                return false;
            }
        }
        return true;
    }
}
```

written by Fanchao original link here

Solution 3

```
    public boolean isReflected(int[][] points) {
     int max = Integer.MIN_VALUE;
     int min = Integer.MAX_VALUE;
     HashSet<String> set = new HashSet<>();
     for(int[] p:points){
         max = Math.max(max,p[0]);
         min = Math.min(min,p[0]);
         String str = p[0] + "a" + p[1];
         set.add(str);
     }
     int sum = max+min;
     for(int[] p:points){
         //int[] arr = {sum-p[0],p[1]};
         String str = (sum-p[0]) + "a" + p[1];
         if( !set.contains(str))
             return false;

     }
     return true;
 }
```

written by juanren original link here

From Leetcoder.

Count Numbers with Unique Digits

Given a **non-negative** integer n, count all numbers with unique digits, x, where $0 \le x < 10^n$.

Example:
Given n = 2, return 91. (The answer should be the total numbers in the range of $0 \le x < 100$, excluding `[11,22,33,44,55,66,77,88,99]`)

1. A direct way is to use the backtracking approach.
2. Backtracking should contains three states which are (the current number, number of steps to get that number and a bitmask which represent which number is marked as visited so far in the current number). Start with state (0,0,0) and count all valid number till we reach number of steps equals to 10^n.
3. This problem can also be solved using a dynamic programming approach and some knowledge of combinatorics.
4. Let f(k) = count of numbers with unique digits with length equals k.
5. f(1) = 10, ..., f(k) = 9 * 9 * 8 * ... (9 - k + 2) [The first factor is 9 because a number cannot start with 0].

Credits:
Special thanks to @memoryless for adding this problem and creating all test cases.

Solution 1

Following the hint. Let f(n) = count of number with unique digits of length n.

f(1) = 10. (0, 1, 2, 3, ..., 9)

f(2) = 9 * 9. Because for each number i from 1, ..., 9, we can pick j to form a 2-digit number ij and there are 9 numbers that are different from i for j to choose from.

f(3) = f(2) * 8 = 9 * 9 * 8. Because for each number with unique digits of length 2, say ij, we can pick k to form a 3 digit number ijk and there are 8 numbers that are different from i and j for k to choose from.

Similarly f(4) = f(3) * 7 = 9 * 9 * 8 * 7....

...

f(10) = 9 * 9 * 8 * 7 * 6 * ... * 1

f(11) = 0 = f(12) = f(13)....

any number with length > 10 couldn't be unique digits number.

The problem is asking for numbers from 0 to 10^n. Hence return f(1) + f(2) + .. + f(n)

As @4acreg suggests, There are only 11 different ans. You can create a lookup table for it. This problem is O(1) in essence.

```
public int countNumbersWithUniqueDigits(int n) {
      if (n == 0)     return 1;

      int res = 10;
      int uniqueDigits = 9;
      int availableNumber = 9;
      while (n-- > 1 && availableNumber > 0) {
          uniqueDigits = uniqueDigits * availableNumber;
          res += uniqueDigits;
          availableNumber--;
      }
      return res;
  }
```

written by stupidbird911 original link here

Solution 2

The idea is to append one digit at a time recursively (only append digits that has not been appended before). Number zero is a special case, because we don't want to deal with the leading zero, so it is counted separately at the beginning of the program. The running time for this program is O(10!) worst case, or O(n!) if n < 10.

The OJ gives wrong answer when n = 0 and n = 1. The correct answer should be:

> 0, 1
>
> 1, 10
>
> 2, 91
>
> 3, 739
>
> 4, 5275
>
> 5, 32491
>
> 6, 168571
>
> 7, 712891
>
> 8, 2345851
>
> 9, 5611771
>
> 10 and beyond, 8877691

```java
public class Solution {
    public static int countNumbersWithUniqueDigits(int n) {
        if (n > 10) {
            return countNumbersWithUniqueDigits(10);
        }
        int count = 1; // x == 0
        long max = (long) Math.pow(10, n);

        boolean[] used = new boolean[10];

        for (int i = 1; i < 10; i++) {
            used[i] = true;
            count += search(i, max, used);
            used[i] = false;
        }

        return count;
    }

    private static int search(long prev, long max, boolean[] used) {
        int count = 0;
        if (prev < max) {
            count += 1;
        } else {
            return count;
        }

        for (int i = 0; i < 10; i++) {
            if (!used[i]) {
                used[i] = true;
                long cur = 10 * prev + i;
                count += search(cur, max, used);
                used[i] = false;
            }
        }

        return count;
    }
}
```

written by lzb700m original link here

Solution 3

test case is wrong when n==1, it should be 11 instead of 10, from 0 to 10 inclusively

n=2 10+9 * 9

n=3 10+9 * 9+9 * 9 * 8

n=4 10+9 * 9+9 * 9 * 8+9 * 9 * 8 * 7

...

when n > 10, the total number won't increase any more, so we set n=10 in that case

```
public class Solution {
    public int countNumbersWithUniqueDigits(int n) {
        if (n == 0) {
            return 2;
        }
        if (n == 1) {
            return 10; // should be 11
        }
        n = Math.min(n, 10);
        int sum = 10;
        int tmp = 9;
        for (int i = 1; i < n; i++) {
            tmp *= 10 - i;
            sum += tmp;
        }
        return sum;
    }
}
```

written by xuyirui original link here

From Leetcoder.

Rearrange String k Distance Apart

Given a non-empty string **str** and an integer **k**, rearrange the string such that the same characters are at least distance **k** from each other.

All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string `""`.

Example 1:

```
str = "aabbcc", k = 3

Result: "abcabc"

The same letters are at least distance 3 from each other.
```

Example 2:

```
str = "aaabc", k = 3

Answer: ""

It is not possible to rearrange the string.
```

Example 3:

```
str = "aaadbbcc", k = 2

Answer: "abacabcd"

Another possible answer is: "abcabcda"

The same letters are at least distance 2 from each other.
```

Credits:

Special thanks to @elmirap for adding this problem and creating all test cases.

Solution 1

key point: using cache during processing heap data.

new version:

```
class Solution {
public:
    string rearrangeString(string str, int k) {
        if(k == 0) return str;
        int length = (int)str.size();

        string res;
        unordered_map<char, int> dict;
        priority_queue<pair<int, char>> pq;

        for(char ch : str) dict[ch]++;
        for(auto it = dict.begin(); it != dict.end(); it++){
            pq.push(make_pair(it->second, it->first));
        }

        while(!pq.empty()){
            vector<pair<int, char>> cache; //store used char during one while loo
p
            int count = min(k, length); //count: how many steps in a while loop
            for(int i = 0; i < count; i++){
                if(pq.empty()) return "";
                auto tmp = pq.top();
                pq.pop();
                res.push_back(tmp.second);
                if(--tmp.first > 0) cache.push_back(tmp);
                length--;
            }
            for(auto p : cache) pq.push(p);
        }
        return res;
    }
};
```

old version:

```
class Solution {
    struct mycompare{
        bool operator()(pair<int, char>& p1, pair<int, char>& p2){
            if(p1.first == p2.first) return p1.second > p2.second;
            return p1.first < p2.first;
        }
    };
public:
    string rearrangeString(string str, int k) {
        if(k == 0) return str;
        unordered_map<char, int> dict;
        for(char ch : str) dict[ch]++;
        int left = (int)str.size();
        priority_queue<pair<int, char>, vector<pair<int, char>>, mycompare > pq;
        for(auto it = dict.begin(); it != dict.end(); it++){
            pq.push(make_pair(it->second, it->first));
        }
        string res;

        while(!pq.empty()){
            vector<pair<int, char>> cache;
            int count = min(k, left);
            for(int i = 0; i < count; i++){
                if(pq.empty()) return "";
                auto tmp = pq.top();
                pq.pop();
                res.push_back(tmp.second);
                if(--tmp.first > 0) cache.push_back(tmp);
                left--;
            }
            for(auto p : cache){
                pq.push(p);
            }
        }
        return res;
    }
};
```

written by sxycwzwzq original link here

Solution 2

This is a greedy problem.
Every time we want to find the best candidate: which is the character with the largest remaining count. Thus we will be having two arrays.
One count array to store the remaining count of every character. Another array to keep track of the most left position that one character can appear. We will iterated through these two array to find the best candidate for every position. Since the array is fixed size, it will take constant time to do this.
After we find the candidate, we update two arrays.

```
public class Solution {
    public String rearrangeString(String str, int k) {
        int length = str.length();
        int[] count = new int[26];
        int[] valid = new int[26];
        for(int i=0;i<length;i++){
            count[str.charAt(i)-'a']++;
        }
        StringBuilder sb = new StringBuilder();
        for(int index = 0;index<length;index++){
            int candidatePos = findValidMax(count, valid, index);
            if( candidatePos == -1) return "";
            count[candidatePos]--;
            valid[candidatePos] = index+k;
            sb.append((char)('a'+candidatePos));
        }
        return sb.toString();
    }

   private int findValidMax(int[] count, int[] valid, int index){
       int max = Integer.MIN_VALUE;
       int candidatePos = -1;
       for(int i=0;i<count.length;i++){
           if(count[i]>0 && count[i]>max && index>=valid[i]){
               max = count[i];
               candidatePos = i;
           }
       }
       return candidatePos;
   }
}
```

At first I was considering using PriorityQueue and referring to this post:
c++ unordered*map priority*queue solution using cache
I have doubts for this solution. If we have "abba", k=2; It seems we might end up with "abba" as the result. Since in the second while loop, I'm not sure 'a' or 'b' will be polled out first.
In Java, when two keys in PriorityQueue have same value, there is no guarantee on the order poll() will return. But I'm not sure how heap is implemented in C++.

written by OrlandoCheno308 original link here

Solution 3

```
import java.util.SortedSet;
import java.util.TreeSet;
public class Solution {
    public String rearrangeString(String str, int k) {
        if(k < 2) return str;
      int[] times = new int[26];
      for(int i = 0; i < str.length(); i++){
          ++times[str.charAt(i) - 'a'];
      }
      SortedSet<int[]> set = new TreeSet<int[]>(new Comparator<int[]>(){
          @Override
          public int compare(int[] a, int[] b){
              return a[0] == b[0] ? Integer.compare(a[1], b[1]) : Integer.compare
(b[0], a[0]);
          }
      });
      for(int i = 0; i < 26; i++){
          if(times[i] != 0){
            set.add(new int[]{times[i], i});
          }
      }
      int cycles = 0;
      int cur = cycles;
      Iterator<int[]> iter = set.iterator();
      char[] res = new char[str.length()];
      while(iter.hasNext()){
          int[] e = iter.next();
          for(int i = 0; i < e[0]; i++){
              res[cur] = (char)('a'+e[1]);
              if(cur > 0 && res[cur] == res[cur-1])
                return "";
              cur += k;
              if(cur >= str.length()){
                  cur = ++cycles;
              }
          }
      }
      return new String(res);
    }
}
```

written by fatalme original link here

Logger Rate Limiter

Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is **not printed in the last 10 seconds**.

Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp, otherwise returns false.

It is possible that several messages arrive roughly at the same time.

Example:

```
Logger logger = new Logger();

// logging string "foo" at timestamp 1
logger.shouldPrintMessage(1, "foo"); returns true;

// logging string "bar" at timestamp 2
logger.shouldPrintMessage(2,"bar"); returns true;

// logging string "foo" at timestamp 3
logger.shouldPrintMessage(3,"foo"); returns false;

// logging string "bar" at timestamp 8
logger.shouldPrintMessage(8,"bar"); returns false;

// logging string "foo" at timestamp 10
logger.shouldPrintMessage(10,"foo"); returns false;

// logging string "foo" at timestamp 11
logger.shouldPrintMessage(11,"foo"); returns true;
```

Credits:
Special thanks to @memoryless for adding this problem and creating all test cases.

Solution 1

Instead of logging print times, I store when it's ok for a message to be printed again. Should be slightly faster, because I don't always have to add or subtract (e.g., `timestamp < log[message] + 10`) but only do in the `true` case. Also, it leads to a shorter/simpler longest line of code. Finally, C++ has 0 as default, so I can just use `ok[message]`.

C++

```
class Logger {
public:

    map<string, int> ok;

    bool shouldPrintMessage(int timestamp, string message) {
        if (timestamp < ok[message])
            return false;
        ok[message] = timestamp + 10;
        return true;
    }
};
```

Python

```
class Logger(object):

    def __init__(self):
        self.ok = {}

    def shouldPrintMessage(self, timestamp, message):
        if timestamp < self.ok.get(message, 0):
            return False
        self.ok[message] = timestamp + 10
        return True
```

Java

```java
public class Logger {

    private Map<String, Integer> ok = new HashMap<>();

    public boolean shouldPrintMessage(int timestamp, String message) {
        if (timestamp < ok.getOrDefault(message, 0))
            return false;
        ok.put(message, timestamp + 10);
        return true;
    }
}
```

written by StefanPochmann original link here

Solution 2

```
import java.util.concurrent.*;

public class Logger {
    ConcurrentHashMap<String, Integer> lastPrintTime;

    /** Initialize your data structure here. */
    public Logger() {
        lastPrintTime = new ConcurrentHashMap<String, Integer>();
    }

    /** Returns true if the message should be printed in the given timestamp, oth
erwise returns false. The timestamp is in seconds granularity. */
    public boolean shouldPrintMessage(int timestamp, String message) {
        Integer last = lastPrintTime.get(message);

        return last == null && lastPrintTime.putIfAbsent(message, timestamp) == n
ull
                || last != null && timestamp - last >= 10 && lastPrintTime.replac
e(message, last, timestamp);

    }
}
```

written by lop original link here

Solution 3

```
 public class Logger {
HashMap<String,Integer> map;
/** Initialize your data structure here. */
public Logger() {
    map=new HashMap<>();
}

/** Returns true if the message should be printed in the given timestamp, otherwis
e returns false. The timestamp is in seconds granularity. */
public boolean shouldPrintMessage(int timestamp, String message) {
//update timestamp of the message if the message is coming in for the first time,o
r the last coming time is earlier than 10 seconds from now
    if(!map.containsKey(message)||timestamp-map.get(message)>=10){
        map.put(message,timestamp);
        return true;
    }
    return false;
}
```

}

written by bxia00801 original link here

From Leetcoder.

Sort Transformed Array

Given a **sorted** array of integers *nums* and integer values a, b and c. Apply a function of the form $f(x) = ax^2 + bx + c$ to each element x in the array.

The returned array must be in **sorted order**.

Expected time complexity: **O(*n*)**

Example:

```
nums = [-4, -2, 2, 4], a = 1, b = 3, c = 5,

Result: [3, 9, 15, 33]

nums = [-4, -2, 2, 4], a = -1, b = 3, c = 5

Result: [-23, -5, 1, 7]
```

Credits:
Special thanks to @elmirap for adding this problem and creating all test cases.

Solution 1

the problem seems to have many cases a>0, a=0,a<0, (when a=0, b>0, b<0). However, they can be combined into just 2 cases: a>0 or a<0

1.a>0, two ends in original array are bigger than center if you learned middle school math before.

2.a<0, center is bigger than two ends.

so use two pointers i, j and do a merge-sort like process. depending on sign of a, you may want to start from the beginning or end of the transformed array. For a==0 case, it does not matter what b's sign is. The function is monotonically increasing or decreasing. you can start with either beginning or end.

```
public class Solution {
    public int[] sortTransformedArray(int[] nums, int a, int b, int c) {
        int n = nums.length;
        int[] sorted = new int[n];
        int i = 0, j = n - 1;
        int index = a >= 0 ? n - 1 : 0;
        while (i <= j) {
            if (a >= 0) {
                sorted[index--] = quad(nums[i], a, b, c) >= quad(nums[j], a, b, c)
? quad(nums[i++], a, b, c) : quad(nums[j--], a, b, c);
            } else {
                sorted[index++] = quad(nums[i], a, b, c) >= quad(nums[j], a, b, c)
? quad(nums[j--], a, b, c) : quad(nums[i++], a, b, c);
            }
        }
        return sorted;
    }

    private int quad(int x, int a, int b, int c) {
        return a * x * x + b * x + c;
    }
}
```

written by xuyirui original link here

Solution 2

$ax^2+bx+c = a(x + b/2a)^2 + c - b^2/4a$

So use offset as **$c - b^2/4a$**

ax^2+bx+c - offset will be always positive or negative

Then iteratively pop max (or min if a is negative) from both ends and push it into final array

```
function sortTransformedArray(nums, a, b, c) {
    var arr = nums.map(n => a * n * n + b * n + c);
    var offset = a ? c - (b * b) / (4 * a) : Math.min(arr[0], arr.slice(-1)[0]);
    var res = [];

    for (var l = 0, r = arr.length - 1; l <= r;) {
        if (Math.abs(arr[l] - offset) >= Math.abs(arr[r] - offset)) {
            res.push(arr[l++]);
        } else {
            res.push(arr[r--]);
        }
    }

    return res[0] > res[res.length - 1] ? res.reverse() : res;
}
```

written by linfongi original link here

Solution 3

```
class Solution {
    public:
        vector<int> sortTransformedArray(vector<int>& nums, int a, int b, int c)
{
            vector<int> res(nums.size());
            if (nums.size() == 0) return res;
            int i = 0, j = nums.size() - 1;
            if (a > 0) {
                int index = nums.size() - 1;
                while (i <= j) {
                    if (transform(nums[i], a, b, c) > transform(nums[j], a, b, c)
) {
                        res[index--] = transform(nums[i], a, b, c);
                        i++;
                    } else {
                        res[index--] = transform(nums[j], a, b, c);
                        j--;
                    }
                }
            } else {
                int index = 0;
                while (i <= j) {
                    if (transform(nums[i], a, b, c) < transform(nums[j], a, b, c)
) {
                        res[index++] = transform(nums[i], a, b, c);
                        i++;
                    } else {
                        res[index++] = transform(nums[j], a, b, c);
                        j--;
                    }
                }
            }
            return res;
        }

        int transform(int num, int a, int b, int c) {
            return a * num * num + b * num + c;
        }
};
```

written by xiaohui5319 original link here

From Leetcoder.

Bomb Enemy

Given a 2D grid, each cell is either a wall `'W'`, an enemy `'E'` or empty `'0'` (the number zero), return the maximum enemies you can kill using one bomb.
The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is too strong to be destroyed.
Note that you can only put the bomb at an empty cell.

Example:

```
For the given grid

0 E 0 0
E 0 W E
0 E 0 0

return 3. (Placing a bomb at (1,1) kills 3 enemies)
```

Credits:
Special thanks to @memoryless for adding this problem and creating all test cases.

Solution 1

Walk through the matrix. At the start of each non-wall-streak (row-wise or column-wise), count the number of hits in that streak and remember it. O(mn) time, O(n) space.

```
int maxKilledEnemies(vector<vector<char>>& grid) {
    int m = grid.size(), n = m ? grid[0].size() : 0;
    int result = 0, rowhits, colhits[n];
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            if (!j || grid[i][j-1] == 'W') {
                rowhits = 0;
                for (int k=j; k<n && grid[i][k] != 'W'; k++)
                    rowhits += grid[i][k] == 'E';
            }
            if (!i || grid[i-1][j] == 'W') {
                colhits[j] = 0;
                for (int k=i; k<m && grid[k][j] != 'W'; k++)
                    colhits[j] += grid[k][j] == 'E';
            }
            if (grid[i][j] == '0')
                result = max(result, rowhits + colhits[j]);
        }
    }
    return result;
}
```

written by StefanPochmann original link here

Solution 2

Pretty straight forward O(mn2) solution. Cacheing the last search result could save a lot of time. Welcome to discuss.

```
public class Solution {
public int maxKilledEnemies(char[][] grid) {
    if (grid == null || grid.length == 0 || grid[0].length == 0) {
        return 0;
    }
    int ret = 0;
    int row = grid.length;
    int col = grid[0].length;
    int rowCache = 0;
    int[] colCache = new int[col];
    for (int i = 0;i < row; i++) {
        for (int j = 0; j < col; j++) {
            if (j == 0 || grid[i][j-1] == 'W') {
                rowCache = 0;
                for (int k = j; k < col && grid[i][k] != 'W'; k++) {
                    rowCache += grid[i][k] == 'E' ? 1 : 0;
                }
            }
            if (i == 0 || grid[i-1][j] == 'W') {
                colCache[j] = 0;
                for (int k = i;k < row && grid[k][j] != 'W'; k++) {
                    colCache[j] += grid[k][j] == 'E' ? 1 :0;
                }
            }
            if (grid[i][j] == '0') {
                ret = Math.max(ret, rowCache + colCache[j]);
            }
        }
    }
    return ret;
}
```

}

written by francis.yang1991 original link here

Solution 3

Compute a matrix for row-wise hits and one for column-wise hits. Then find the maximum.

```
def maxKilledEnemies(self, grid):
    def hits(grid):
        return [[h
                 for block in ''.join(row).split('W')
                 for h in [block.count('E')] * len(block) + [0]]
                for row in grid]
    rowhits = hits(grid)
    colhits = zip(*hits(zip(*grid)))
    return max([rh + ch
                for row in zip(grid, rowhits, colhits)
                for cell, rh, ch in zip(*row)
                if cell == '0'] or [0])
```

written by StefanPochmann original link here

From Leetcoder.

Design Hit Counter

Design a hit counter which counts the number of hits received in the past 5 minutes.

Each function accepts a timestamp parameter (in seconds granularity) and you may assume that calls are being made to the system in chronological order (ie, the timestamp is monotonically increasing). You may assume that the earliest timestamp starts at 1.

It is possible that several hits arrive roughly at the same time.

Example:

```
HitCounter counter = new HitCounter();

// hit at timestamp 1.
counter.hit(1);

// hit at timestamp 2.
counter.hit(2);

// hit at timestamp 3.
counter.hit(3);

// get hits at timestamp 4, should return 3.
counter.getHits(4);

// hit at timestamp 300.
counter.hit(300);

// get hits at timestamp 300, should return 4.
counter.getHits(300);

// get hits at timestamp 301, should return 3.
counter.getHits(301);
```

Follow up:
What if the number of hits per second could be very large? Does your design scale?

Credits:
Special thanks to @elmirap for adding this problem and creating all test cases.

Solution 1

O(s) s is total seconds in given time interval, in this case 300. basic ideal is using buckets. 1 bucket for every second because we only need to keep the recent hits info for 300 seconds. hit[] array is wrapped around by mod operation. Each hit bucket is associated with times[] bucket which record current time. If it is not current time, it means it is 300s or 600s... ago and need to reset to 1.

```
public class HitCounter {
    private int[] times;
    private int[] hits;
    /** Initialize your data structure here. */
    public HitCounter() {
        times = new int[300];
        hits = new int[300];
    }

    /** Record a hit.
        @param timestamp - The current timestamp (in seconds granularity). */
    public void hit(int timestamp) {
        int index = timestamp % 300;
        if (times[index] != timestamp) {
            times[index] = timestamp;
            hits[index] = 1;
        } else {
            hits[index]++;
        }
    }

    /** Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds granularity). */
    public int getHits(int timestamp) {
        int total = 0;
        for (int i = 0; i < 300; i++) {
            if (timestamp - times[i] < 300) {
                total += hits[i];
            }
        }
        return total;
    }
}
```

written by xuyirui original link here

Solution 2

In this problem, I use a queue to record the information of all the hits. Each time we call the function getHits( ), we have to delete the elements which hits beyond 5 mins (300). The result would be the length of the queue :)

```
public class HitCounter {
        Queue<Integer> q = null;
        /** Initialize your data structure here. */
        public HitCounter() {
            q = new LinkedList<Integer>();
        }

        /** Record a hit.
            @param timestamp - The current timestamp (in seconds granularity). */
        public void hit(int timestamp) {
            q.offer(timestamp);
        }

        /** Return the number of hits in the past 5 minutes.
            @param timestamp - The current timestamp (in seconds granularity). */
        public int getHits(int timestamp) {
            while(!q.isEmpty() && timestamp - q.peek() >= 300) {
                q.poll();
            }
            return q.size();
        }
    }
```

written by cyqz1993 original link here

Solution 3

This solution is based on the idea in this post: https://nuttynanaus.wordpress.com/2014/03/09/software-engineer-interview-questions/
There are two solutions, the first one we choose 1s as granularity, the other is full accuracy(see the post). We call move() before hit() and getHits(). move() will take time at most O(N), where N is the length of the array.

```
public class HitCounter {
    int N;
    int[] count;
    int lastPosition;
    int lastTime;
    int sum;

    /** Initialize your data structure here. */
    public HitCounter() {
        N = 300;
        count = new int[N];
        lastPosition = 0;
        lastTime = 0;
        sum = 0;
    }

    /** Record a hit.
        @param timestamp - The current timestamp (in seconds granularity). */
    public void hit(int timestamp) {
        move(timestamp);
        count[lastPosition]++;
        sum++;
    }

    /** Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds granularity). */
    public int getHits(int timestamp) {
        move(timestamp);
        return sum;
    }

    void move(int timestamp){
        int gap = Math.min(timestamp-lastTime, N);
        for(int i=0; i<gap;i++){
            lastPosition = (lastPosition+1)%N;
            sum -= count[lastPosition];
            count[lastPosition] = 0;
        }
        lastTime = timestamp;
    }
}
```

written by OrlandoChen0308 original link here

Max Sum of Rectangle No Larger Than K

Given a non-empty 2D matrix *matrix* and an integer *k*, find the max sum of a rectangle in the *matrix* such that its sum is no larger than*k*.

Example:

```
Given matrix = [
  [1,  0, 1],
  [0, -2, 3]
]
k = 2
```

The answer is `2`. Because the sum of rectangle `[[0, 1], [-2, 3]]` is 2 and 2 is the max number no larger than k (k = 2).

Note:

1. The rectangle inside the matrix must have an area > 0.
2. What if the number of rows is much larger than the number of columns?

Credits:
Special thanks to @fujiaozhu for adding this problem and creating all test cases.

Solution 1

The naive solution is brute-force, which is O((mn)^2). In order to be more efficient, I tried something similar to Kadane's algorithm. The only difference is that here we have upper bound restriction K. Here's the easily understanding video link for the problem "find the max sum rectangle in 2D array": Maximum Sum Rectangular Submatrix in Matrix dynamic programming/2D kadane (Trust me, it's really easy and straightforward).

Once you are clear how to solve the above problem, the next step is to find the max sum no more than K in an array. This can be done within O(nlogn), and you can refer to this article: max subarray sum no more than k.

For the solution below, I assume that the number of rows is larger than the number of columns. Thus in general time complexity is O[min(m,n)^2 * max(m,n) * log(max(m,n))], space O(max(m, n)).

```
int maxSumSubmatrix(vector<vector<int>>& matrix, int k) {
    if (matrix.empty()) return 0;
    int row = matrix.size(), col = matrix[0].size(), res = INT_MIN;
    for (int l = 0; l < col; ++l) {
        vector<int> sums(row, 0);
        for (int r = l; r < col; ++r) {
            for (int i = 0; i < row; ++i) {
                sums[i] += matrix[i][r];
            }

            // Find the max subarray no more than K
            set<int> accuSet;
            accuSet.insert(0);
            int curSum = 0, curMax = INT_MIN;
            for (int sum : sums) {
                curSum += sum;
                set<int>::iterator it = accuSet.lower_bound(curSum - k);
                if (it != accuSet.end()) curMax = std::max(curMax, curSum - *it);
                accuSet.insert(curSum);
            }
            res = std::max(res, curMax);
        }
    }
    return res;
}
```

written by java-chao original link here

Solution 2

```
/* first  consider the situation matrix is 1D
    we can save every sum of 0~i(0<=i<len) and binary search previous sum to find
    possible result for every index, time complexity is O(NlogN).
    so in 2D matrix, we can sum up all values from row i to row j and create a 1D
array
    to use 1D array solution.
    If col number is less than row number, we can sum up all values from col i to
col j
    then use 1D array solution.
*/
public int maxSumSubmatrix(int[][] matrix, int target) {
    int row = matrix.length;
    if(row==0)return 0;
    int col = matrix[0].length;
    int m = Math.min(row,col);
    int n = Math.max(row,col);
    //indicating sum up in every row or every column
    boolean colIsBig = col>row;
    int res = Integer.MIN_VALUE;
    for(int i = 0;i<m;i++){
        int[] array = new int[n];
        // sum from row j to row i
        for(int j = i;j>=0;j--){
            int val = 0;
            TreeSet<Integer> set = new TreeSet<Integer>();
            set.add(0);
            //traverse every column/row and sum up
            for(int k = 0;k<n;k++){
                array[k]=array[k]+(colIsBig?matrix[j][k]:matrix[k][j]);
                val = val + array[k];
                //use  TreeMap to binary search previous sum to get possible resu
lt
                Integer subres = set.ceiling(val-target);
                if(null!=subres){
                    res=Math.max(res,val-subres);
                }
                set.add(val);
            }
        }
    }
    return res;
}
```

written by wuzixigua original link here

Solution 3

I got a TLE for the Python code below, because the time cost of bisect.insort is O(n) for a built-in list.

The code was rejudged as accepted just now, but very slow... 1800ms+

```
class Solution(object):
    def maxSumSubmatrix(self, matrix, k):
        """
        :type matrix: List[List[int]]
        :type k: int
        :rtype: int
        """
        m = len(matrix)
        n = len(matrix[0]) if m else 0

        M = max(m, n)
        N = min(m, n)
        ans = None
        for x in range(N):
            sums = [0] * M
            for y in range(x, N):
                slist, num = [], 0
                for z in range(M):
                    sums[z] += matrix[z][y] if m > n else matrix[y][z]
                    num += sums[z]
                    if num <= k: ans = max(ans, num)
                    i = bisect.bisect_left(slist, num - k)
                    if i != len(slist): ans = max(ans, num - slist[i])
                    bisect.insort(slist, num)
        return ans or 0
```

Could anybody share a more efficient Python solution? Thank you :D

written by %E5%9C%A8%E7%BA%BF%E7%96%AF%E7%8B%82 original link here

From Leetcoder.

Nested List Weight Sum II

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Different from the previous question where weight is increasing from root to leaf, now the weight is defined from bottom up. i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.

Example 1:
Given the list `[[1,1],2,[1,1]]`, return **8**. (four 1's at depth 1, one 2 at depth 2)

Example 2:
Given the list `[1,[4,[6]]]`, return **17**. (one 1 at depth 3, one 4 at depth 2, and one 6 at depth 1; 1*3 + 4*2 + 6*1 = 17)

Solution 1

Inspired by lzb700m's solution and one of mine. Instead of multiplying by depth, add integers multiple times (by going level by level and adding the unweighted sum to the weighted sum after each level).

```
public int depthSumInverse(List<NestedInteger> nestedList) {
    int unweighted = 0, weighted = 0;
    while (!nestedList.isEmpty()) {
        List<NestedInteger> nextLevel = new ArrayList<>();
        for (NestedInteger ni : nestedList) {
            if (ni.isInteger())
                unweighted += ni.getInteger();
            else
                nextLevel.addAll(ni.getList());
        }
        weighted += unweighted;
        nestedList = nextLevel;
    }
    return weighted;
}
```

written by StefanPochmann original link here

Solution 2

The weight increases from the leaf to the root.

However, the following situation is not clearly defined. I will illustrate it using a tree structure.

```
    a
  /     \
 b       c
           \
            d
```

What is the weight of a? Is it 2 to 3?

written by hyj143 original link here

Solution 3

The idea is to pass the current found integer sum into the next level of recursion, and return it back again. So that we don't have to count the number of levels in the nested list.

I think code itself is quite self explanatory.

```
public class Solution {
    public int depthSumInverse(List<NestedInteger> nestedList) {
        return helper(nestedList, 0);
    }

    private int helper(List<NestedInteger> niList, int prev) {
        int intSum = prev;
        List<NestedInteger> levelBreak = new ArrayList<>();

        for (NestedInteger ni : niList) {
            if (ni.isInteger()) {
                intSum += ni.getInteger();
            } else {
                levelBreak.addAll(ni.getList());
            }
        }

        int listSum = levelBreak.isEmpty()? 0 : helper(levelBreak, intSum);

        return listSum + intSum;
    }
}
```

written by lzb700m original link here

From Leetcoder.

Water and Jug Problem

You are given two jugs with capacities x and y litres. There is an infinite amount of water supply available. You need to determine whether it is possible to measure exactly z litres using these two jugs.

If z liters of water is measurable, you must have z liters of water contained within **one or both buckets** by the end.

Operations allowed:

- Fill any of the jugs completely with water.
- Empty any of the jugs.
- Pour water from one jug into another till the other jug is completely full or the first jug itself is empty.

Example 1: (From the famous *"Die Hard"* example)

```
Input: x = 3, y = 5, z = 4
Output: True
```

Example 2:

```
Input: x = 2, y = 6, z = 5
Output: False
```

Credits:
Special thanks to @vinod23 for adding this problem and creating all test cases.

Solution 1

only thing we should proof is this:

```
if `x` and `y` are coprime,
then we can and only can reach every integer `z` in `[0, x + y]`. (1)
```

then for a GCD `g`, from `gx` and `gy`, we can and only can reach every `z` in `{i * g | i in [0, x + y] }`

now, let's see how to proof (1). let x be the less one, and y the greater one. then fill the two jug to full, we have `x` and `y` water each and `x + y` water in total. then we pour out x water each time until we can't.

now we have these different z:

```
y + x,  y,  y - x,  y - 2x, ... , y % x
```

finally we have `y % x` water left, we pour it into the x jug, then fill the y jug to full. now the two jugs have `y % x` and `y` water separately, and `y + y % x` water in total. then we pour from y jug into x jug until the x jug is full, afterwards do the same thing like before, to pour out x water each time until we can't.

finally we get `(y + y % x) % x = (y % x + y % x) % x = (2y) % x` water left.

now we have these different z:

```
y + y % x,   y + y % x - x,  y + y % x - 2x, ... ,  (2y) % x
```

do this x times, we get z:

```
y + (2y) % x,  y + (2y) % x - x, y + (2y) % x - 2x, ..., (3y) % x
:
:
:
y + ((x-1)y) % x,  y + ((x-1)y) % x - x,  y + ((x-1)y) % x - 2x, ... ,  (xy) % x
```

then you see `(xy) % x = 0`, and

```
y % x, (2y) % x, (3y) % x, ... , ((x-1)y) % x just are 1, 2, 3, 4, 5, ... , x -
1. (2)
```

proof for (2): modulo x could get x - 1 different results at most exclusive 0, that's 1,2,3,...,x-1. we have x - 1 expressions, suppose there is two same, let a != b in [1, x-1] and `(ay) % x = (by) % x`, then we get `((a - b)y) % x = 0`, then `((a - b) % x) * (y % x) = 0`, `(a - b) % x = 0`. for `1 <= a, b <= x - 1`, so we get `a = b`. it's impossible.

so we can get any z in [0, x + y] water for coprime x and y.

code:

```
   bool canMeasureWater(int x, int y, int z) {
    return z == 0 || z <= (long long)x + y && z % __gcd(x, y) == 0;
}
```

p.s. the testcases are too weak. you may get overflow without long long, say 2147483647, 234524287, 1. some code get AC, while runs wrong on this 4,6,8.

written by lishq991 original link here

Solution 2

This is a pure Math problem. We need the knowledge of number theory to cover the proof and solution. No idea why microsoft uses this problem in real interview.

The basic idea is to use the property of Bézout's identity and check if z is a multiple of GCD(x, y)

Quote from wiki:

> Bézout's identity (also called Bézout's lemma) is a theorem in the elementary theory of numbers:
>
> let a and b be nonzero integers and let d be their greatest common divisor. Then there exist integers x and y such that ax+by=d
>
> In addition, the greatest common divisor d is the smallest positive integer that can be written as ax + by
>
> every integer of the form ax + by is a multiple of the greatest common divisor d.

If a or b is negative this means we are emptying a jug of x or y gallons respectively.

Similarly if a or b is positive this means we are filling a jug of x or y gallons respectively.

x = 4, y = 6, z = 8.

GCD(4, 6) = 2

8 is multiple of 2

so this input is valid and we have:

-1 * 4 + 6 * 2 = 8

In this case, there is a solution obtained by filling the 6 gallon jug twice and emptying the 4 gallon jug once. (Solution. Fill the 6 gallon jug and empty 4 gallons to the 4 gallon jug. Empty the 4 gallon jug. Now empty the remaining two gallons from the 6 gallon jug to the 4 gallon jug. Next refill the 6 gallon jug. This gives 8 gallons in the end)

See wiki:

Bézout's identity

and comments in the code

```java
public boolean canMeasureWater(int x, int y, int z) {
    //limit brought by the statement that water is finallly in one or both buckets
    if(x + y < z) return false;
    //case x or y is zero
    if( x == z || y == z || x + y == z ) return true;

    //get GCD, then we can use the property of Bézout's identity
    return z%GCD(x, y) == 0;
}

public int GCD(int a, int b){
    while(b != 0 ){
        int temp = b;
        b = a%b;
        a = temp;
    }
    return a;
}
```

written by hpplayer original link here

Solution 3

Why the test case 1 2 5 return false? Because I think 5 = 1 + 2 + 2.

written by fangdanzai original link here

From Leetcoder.

Find Leaves of Binary Tree

Given a binary tree, find all leaves and then remove those leaves. Then repeat the previous steps until the tree is empty.

Example:
Given binary tree

```
          1
         / \
        2   3
       / \
      4   5
```

Returns `[4, 5, 3], [2], [1]`.

Explanation:

1. Remove the leaves `[4, 5, 3]` from the tree

```
          1
         /
        2
```

2. Remove the leaf `[2]` from the tree

```
          1
```

3. Remove the leaf `[1]` from the tree

```
          []
```

Returns `[4, 5, 3], [2], [1]`.

Credits:
Special thanks to @elmirap for adding this problem and creating all test cases.

Solution 1

```
class Solution {
private:
    int dfs(TreeNode* root, vector<vector<int>>& res){
        if(!root) return 0;
        int level = max(dfs(root->left, res), dfs(root->right, res)) + 1;
        if(level > (int)res.size()) res.push_back(vector<int>());
        res[level - 1].push_back(root->val);
        return level;
    }
public:
    vector<vector<int>> findLeaves(TreeNode* root) {
        vector<vector<int>> res;
        dfs(root, res);
        return res;
    }
};
```

written by sxycwzwzq original link here

Solution 2

```
public class Solution {
    private List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> findLeaves(TreeNode root) {
        helper(root);
        return res;
    }
    private int helper(TreeNode node){
        if(null==node){
            return -1;
        }
        int level = 1 + Math.max(helper(node.left), helper(node.right));
        if(res.size()<level+1){
            res.add(new ArrayList<Integer>());
        }
        res.get(level).add(node.val);
        return level;
    }
}
```

written by sky-xu original link here

Solution 3

The essential of problem is not to find the leaves, but group leaves of same level together and also to cut the tree. This is the exact role backtracking plays. The helper function returns the level which is the distance from its furthest subtree leaf to root, which helps to identify which group the root belongs to

```
    public class Solution {
        public List<List<Integer>> findLeaves(TreeNode root) {
            List<List<Integer>> list = new ArrayList<>();
            findLeavesHelper(list, root);
            return list;
        }

  // return the level of root
        private int findLeavesHelper(List<List<Integer>> list, TreeNode root) {
            if (root == null) {
                return -1;
            }
            int leftLevel = findLeavesHelper(list, root.left);
            int rightLevel = findLeavesHelper(list, root.right);
            int level = Math.max(leftLevel, rightLevel) + 1;
            if (list.size() == level) {
                list.add(new ArrayList<>());
            }
            list.get(level).add(root.val);
            root.left = root.right = null;
            return level;
        }
    }
```

written by xuyirui original link here

From Leetcoder.

Valid Perfect Square

Given a positive integer *num*, write a function which returns True if *num* is a perfect square else False.

Note: Do not use any built-in library function such as `sqrt`.

Example 1:

```
Input: 16
Returns: True
```

Example 2:

```
Input: 14
Returns: False
```

Credits:

Special thanks to @elmirap for adding this problem and creating all test cases.

Solution 1

```
public boolean isPerfectSquare(int num) {
    int i = 1;
    while (num > 0) {
        num -= i;
        i += 2;
    }
    return num == 0;
}
```

written by fhqplzj original link here

Solution 2

1. a square number is 1+3+5+7+... Time Complexity O(sqrt(N)) (Credit to lizhibupt, thanks for correcting this).
2. binary search. Time Complexity O(logN)
3. Newton Method. See this wiki page. Time Complexity is close to constant, given a positive integer.

```
public boolean isPerfectSquare(int num) {
  if (num < 1) return false;
  for (int i = 1; num > 0; i += 2)
    num -= i;
  return num == 0;
}

public boolean isPerfectSquare(int num) {
  if (num < 1) return false;
  long left = 1, right = num;// long type to avoid 2147483647 case

  while (left <= right) {
    long mid = left + (right - left) / 2;
    long t = mid * mid;
    if (t > num) {
      right = mid - 1;
    } else if (t < num) {
      left = mid + 1;
    } else {
      return true;
    }
  }

  return false;
}

boolean isPerfectSquare(int num) {
  if (num < 1) return false;
  long t = num / 2;
  while (t * t > num) {
    t = (t + num / t) / 2;
  }
  return t * t == num;
}
```

written by coolguy original link here

Solution 3

Just slightly modified my sqrt solutions. You can find some explanation there.

(Note I renamed the parameter to x because that's the name in the sqrt problem and I like it better.)

Java, C++, C, C#

```
long r = x;
while (r*r > x)
    r = (r + x/r) / 2;
return r*r == x;
```

Python

```
r = x
while r*r > x:
    r = (r + x/r) / 2
return r*r == x
```

Ruby

```
r  = x
r = (r + x/r) / 2 while r*r > x
r*r == x
```

JavaScript

```
r = x;
while (r*r > x)
    r = ((r + x/r) / 2) | 0;
return r*r == x;
```

written by StefanPochmann original link here

From Leetcoder.

Largest Divisible Subset

Given a set of **distinct** positive integers, find the largest subset such that every pair (S_i, S_j) of elements in this subset satisfies: $S_i \% S_j = 0$.

If there are multiple solutions, return any subset is fine.

Example 1:

```
nums: [1,2,3]

Result: [1,2] (of course, [1,3] will also be ok)
```

Example 2:

```
nums: [1,2,4,8]

Result: [1,2,4,8]
```

Credits:

Special thanks to @Stomach_ache for adding this problem and creating all test cases.

Solution 1

The key concept here is:
Given a set of integers that satisfies the property that each pair of integers inside the set are mutually divisible, for a new integer S, S can be placed into the set as long as it can divide the smallest number of the set or is divisible by the largest number of the set.

For example, let's say we have a set P = { 4, 8, 16 }, P satisfies the divisible condition. Now consider a new number 2, it can divide the smallest number 4, so it can be placed into the set; similarly, 32 can be divided by 16, the biggest number in P, it can also placed into P.

Next, let's define:

EDIT: For clarification, the following definitions try to enlarge candidate solutions by appending a larger element at the end of each potential set, while my implementation below is prefixing a smaller element at the front of a set. Conceptually they are equivalent but by adding smaller elements at the front saves the trouble for keeping the correct increasing order for the final answer. Please refer to comments in code for more details.

For an increasingly sorted array of integers a[1 .. n]

T[n] = the length of the largest divisible subset whose largest number is a[n]

T[n+1] = max{ 1 + T[i] if a[n+1] mod a[i] == 0 else 1 }

Now, deducting T[n] becomes straight forward with a DP trick. For the final result we will need to maintain a backtrace array for the answer.

Implementation in C++:

```
class Solution {
public:
    vector<int> largestDivisibleSubset(vector<int>& nums) {
        sort(nums.begin(), nums.end());

        vector<int> T(nums.size(), 0);
        vector<int> parent(nums.size(), 0);

        int m = 0;
        int mi = 0;

        // for(int i = 0; i < nums.size(); ++i) // if extending by larger element
s
        for(int i = nums.size() - 1; i >= 0; --i) // iterate from end to start si
nce it's easier to track the answer index
        {
            // for(int j = i; j >=0; --j) // if extending by larger elements
            for(int j = i; j < nums.size(); ++j)
            {
                // if(nums[i] % nums[j] == 0 && T[i] < 1 + T[j]) // if extending
by larger elements
                // check every a[j] that is larger than a[i]
                if(nums[j] % nums[i] == 0 && T[i] < 1 + T[j])
                {
                    // if a[j] mod a[i] == 0, it means T[j] can form a larger sub
set by putting a[i] into T[j]
                    T[i] = 1 + T[j];
                    parent[i] = j;

                    if(T[i] > m)
                    {
                        m = T[i];
                        mi = i;
                    }
                }
            }
        }

        vector<int> ret;

        for(int i = 0; i < m; ++i)
        {
            ret.push_back(nums[mi]);
            mi = parent[mi];
        }

        // sort(ret.begin(), ret.end()); // if we go by extending larger ends, th
e largest "answer" element will come first since the candidate element we observe
will become larger and larger as i increases in the outermost "for" loop above.
       // alternatively, we can sort nums in decreasing order obviously.

        return ret;
    }
};
```

written by roy14 original link here

Solution 2

```
def largestDivisibleSubset(self, nums):
    S = {-1: set()}
    for x in sorted(nums):
        S[x] = max((S[d] for d in S if x % d == 0), key=len) | {x}
    return list(max(S.values(), key=len))
```

My `S[x]` is the largest subset with `x` as the largest element, i.e., the subset of all divisors of `x` in the input. With `S[-1] = emptyset` as useful base case. Since divisibility is transitive, a multiple `x` of some divisor `d` is also a multiple of all elements in `S[d]`, so it's not necessary to explicitly test divisibility of `x` by all elements in `S[d]`. Testing `x % d` suffices.

While storing entire subsets isn't super efficient, it's also not that bad. To extend a subset, the new element must be divisible by all elements in it, meaning it must be at least twice as large as the largest element in it. So with the 31-bit integers we have here, the largest possible set has size 31 (containing all powers of 2).

written by StefanPochmann original link here

Solution 3

Use DP to track max Set and pre index.

```
public class Solution {
    public List<Integer> largestDivisibleSubset(int[] nums) {
        int n = nums.length;
        int[] count = new int[n];
        int[] pre = new int[n];
        Arrays.sort(nums);
        int max = 0, index = -1;
        for (int i = 0; i < n; i++) {
            count[i] = 1;
            pre[i] = -1;
            for (int j = i - 1; j >= 0; j--) {
                if (nums[i] % nums[j] == 0) {
                    if (1 + count[j] > count[i]) {
                        count[i] = count[j] + 1;
                        pre[i] = j;
                    }
                }
            }
            if (count[i] > max) {
                max = count[i];
                index = i;
            }
        }
        List<Integer> res = new ArrayList<>();
        while (index != -1) {
            res.add(nums[index]);
            index = pre[index];
        }
        return res;
    }
}
```

written by jiangbowei2010 original link here

From Leetcoder.

Plus One Linked List

Given a non-negative number represented as a singly linked list of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Example:

```
Input:
1->2->3

Output:
1->2->4
```

Solution 1

```java
public class Solution {
    public ListNode plusOne(ListNode head) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode i = dummy;
        ListNode j = dummy;

        while (j.next != null) {
            j = j.next;
            if (j.val != 9) {
                i = j;
            }
        }

        if (j.val != 9) {
            j.val++;
        } else {
            i.val++;
            i = i.next;
            while (i != null) {
                i.val = 0;
                i = i.next;
            }
        }

        if (dummy.val == 0) {
            return dummy.next;
        }

        return dummy;
    }
}
```

- i stands for the most significant digit that is going to be incremented if there exists a carry
- dummy node can handle cases such as "9->9>-9" automatically

written by ocean original link here

Solution 2

At the first glance, I want to reverse the inputs, add one, then reverse back. But that is too intuitive and I don't think this is an expected solution. Then what kind of alg would adding one in reverse way for list?

Recursion! With recursion, we can visit list in reverse way! So here is my recursive solution.

```
public ListNode plusOne(ListNode head) {
    if( DFS(head) == 0){
        return head;
    }else{
        ListNode newHead = new ListNode(1);
        newHead.next = head;
        return newHead;
    }
}

public int DFS(ListNode head){
    if(head == null) return 1;

    int carry = DFS(head.next);

    if(carry == 0) return 0;

    int val = head.val + 1;
    head.val = val%10;
    return val/10;
}
```

written by hpplayer original link here

Solution 3

Inspired by others I had to try it myself...

Solution 1, reverse and then increase while reversing back

```
def plusOne(self, head):
    tail = None
    while head:
        head.next, head, tail = tail, head.next, head
    carry = 1
    while tail:
        carry, tail.val = divmod(carry + tail.val, 10)
        if carry and not tail.next:
            tail.next = ListNode(0)
        tail.next, tail, head = head, tail.next, tail
    return head
```

Solution 2, with pure reverse helper

```
def plusOne(self, head):
    def reverse(head):
        rev = None
        while head:
            head.next, head, rev = rev, head.next, head
        return rev
    head = node = reverse(head)
    while node.val == 9:
        node.val = 0
        node.next = node = node.next or ListNode(0)
    node.val += 1
    return reverse(head)
```

If you don't like that bottom while-loop, here's a more normal way I guess:

```
    while node.val == 9:
        node.val = 0
        if not node.next:
            node.next = ListNode(0)
        node = node.next
```

written by StefanPochmann original link here

From Leetcoder.

Range Addition

Assume you have an array of length ***n*** initialized with all **0**'s and are given ***k*** update operations.

Each operation is represented as a triplet: **[startIndex, endIndex, inc]** which increments each element of subarray **A[startIndex ... endIndex]** (startIndex and endIndex inclusive) with **inc**.

Return the modified array after all ***k*** operations were executed.

Example:

```
Given:

    length = 5,
    updates = [
        [1,  3,  2],
        [2,  4,  3],
        [0,  2, -2]
    ]

Output:

    [-2, 0, 3, 5, 3]
```

Explanation:

```
Initial state:
[ 0, 0, 0, 0, 0 ]

After applying operation [1, 3, 2]:
[ 0, 2, 2, 2, 0 ]

After applying operation [2, 4, 3]:
[ 0, 2, 5, 5, 3 ]

After applying operation [0, 2, -2]:
[-2, 0, 3, 5, 3 ]
```

1. Thinking of using advanced data structures? You are thinking it too complicated.
2. For each update operation, do you really need to update all elements between i and j?
3. Update only the first and end element is sufficient.
4. The optimal time complexity is O(***k*** + ***n***) and uses O(1) extra space.

Credits:
Special thanks to @vinod23 for adding this problem and creating all test cases.

Solution 1

Just store every start index for each value and at end index plus one minus it

for example it will look like:

[1 , 3 , 2] , [2, 3, 3] (length = 5)

res[0, 2, ,0, 0 -2]

res[0 ,2, 3, 0, -5]

sum 0, 2, 5, 5, 0

res[0, 2, 5, 5, 0]

```
public int[] getModifiedArray(int length, int[][] updates) {

    int[] res = new int[length];
     for(int[] update : updates) {
        int value = update[2];
        int start = update[0];
        int end = update[1];

        res[start] += value;

        if(end < length - 1)
            res[end + 1] -= value;

    }

    int sum = 0;
    for(int i = 0; i < length; i++) {
        sum += res[i];
        res[i] = sum;
    }

    return res;
}
```

written by lid004 original link here

Solution 2

From the hint, we only need to update first and end element, so we update the startIndex with inc, then update endIndex + 1 with -inc.
Using the example in the problem, We get vector nums = {-2, 2, 3, 2, -2, -3}, then we compute range sum (Range Sum Query - Immutable), that is the final result = {-2, 0, 3, 5, 3}.

```
class Solution {
public:
    vector<int> getModifiedArray(int length, vector<vector<int>>& updates) {
        vector<int> res, nums(length + 1, 0);
        for (int i = 0; i < updates.size(); ++i) {
            nums[updates[i][0]] += updates[i][2];
            nums[updates[i][1] + 1] -= updates[i][2];
        }
        int sum = 0;
        for (int i = 0; i < length; ++i) {
            sum += nums[i];
            res.push_back(sum);
        }
        return res;
    }
};
```

written by grandyang original link here

Solution 3

Just use the first position and the **last + 1** position. The last + 1 position add the opposite value. So when you add the numbers from start to end you will have correct result.

written by willcomeback original link here

From Leetcoder.

Sum of Two Integers

Calculate the sum of two integers a and b, but you are **not allowed** to use the operator `+` and `-`.

Example:
Given $a = 1$ and $b = 2$, return 3.

Credits:
Special thanks to @fujiaozhu for adding this problem and creating all test cases.

Solution 1

I have been confused about bit manipulation for a very long time. So I decide to do a summary about it here.

"&" AND operation, for example, 2 (0010) & 7 (0111) => 2 (0010)

"^" XOR operation, for example, 2 (0010) ^ 7 (0111) => 5 (0101)

"~" NOT operation, for example, ~2(0010) => -3 (1101) what??? Don't get frustrated here. It's called two's complement.

1111 is -1, in two's complement

1110 is -2, which is ~2 + 1, ~0010 => 1101, 1101 + 1 = 1110 => 2

1101 is -3, which is ~3 + 1

so if you want to get a negative number, you can simply do ~x + 1

Reference:

https://en.wikipedia.org/wiki/Two%27s_complement

https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html

For this, problem, for example, we have a = 1, b = 3,

In bit representation, a = 0001, b = 0011,

First, we can use "and"("&") operation between a and b to find a carry.

carry = a & b, then carry = 0001

Second, we can use "xor" ("^") operation between a and b to find the different bit, and assign it to a,

Then, we shift carry one position left and assign it to b, b = 0010.

Iterate until there is no carry (or b == 0)

```java
// Iterative
public int getSum(int a, int b) {
 if (a == 0) return b;
 if (b == 0) return a;

 while (b != 0) {
  int carry = a & b;
  a = a ^ b;
  b = carry << 1;
 }

 return a;
}

// Iterative
public int getSubtract(int a, int b) {
 while (b != 0) {
  int borrow = (~a) & b;
  a = a ^ b;
  b = borrow << 1;
 }

 return a;
}

// Recursive
public int getSum(int a, int b) {
 return (b == 0) ? a : getSum(a ^ b, (a & b) << 1);
}

// Recursive
public int getSubtract(int a, int b) {
 return (b == 0) ? a : getSubtract(a ^ b, (~a & b) << 1);
}

// Get negative number
public int negate(int x) {
 return ~x + 1;
}
```

written by zhaolz original link here

Solution 2

test cases all pass

0 ms

```
public int getSum(int a, int b) {
  if(b == 0){//没有进为的时候完成运算
      return a;
 }
 int sum,carry;
 sum = a^b;//完成第一步加发的运算
 carry = (a&b)<<1;//完成第二步进位并且左移运算
 return getSum(sum,carry);//
 }
```

written by lid004 original link here

Solution 3

```
class Solution {
public:
    int getSum(int a, int b) {
        int sum = a;

        while (b != 0)
        {
            sum = a ^ b;//calculate sum of a and b without thinking the carry
            b = (a & b) << 1;//calculate the carry
            a = sum;//add sum(without carry) and carry
        }

        return sum;
    }
};
```

written by vdvvdd original link here

From Leetcoder.

Super Pow

Your task is to calculate a^b mod 1337 where a is a positive integer and b is an extremely large positive integer given in the form of an array.

Example1:

```
a = 2
b = [3]

Result: 8
```

Example2:

```
a = 2
b = [1,0]

Result: 1024
```

Solution 1

One knowledge: a*b % k = (a%k)*(b%k)%k
Since the power here is an array, we'd better handle it digit by digit.
One observation:
a^1234567 % k = (a^1234560 % k) * (a^7 % k) % k = (a^123456 % k)^10 % k * (a^7 % k) % k
Looks complicated? Let me put it other way:
Suppose f(a, b) calculates a^b % k; Then translate above formula to using f :
f(a,1234567) = f(a, 1234560) * f(a, 7) % k = f(f(a, 123456),10) * f(a,7)%k;
Implementation of this idea:

```
class Solution {
    const int base = 1337;
    int powmod(int a, int k) //a^k mod 1337 where 0 <= k <= 10
    {
        a %= base;
        int result = 1;
        for (int i = 0; i < k; ++i)
            result = (result * a) % base;
        return result;
    }
public:
    int superPow(int a, vector<int>& b) {
        if (b.empty()) return 1;
        int last_digit = b.back();
        b.pop_back();
        return powmod(superPow(a, b), 10) * powmod(a, last_digit) % base;
    }
};
```

Note: This approach is definitely not the fastest, but it is something one can quickly understand and write out when asked in an interview.
And this approach is not using any built-in "pow" functions, I think this is also what the interviewer would expect you to do.
Hope it helps!

written by fentoyal original link here

Solution 2

1337 only has two divisors 7 and 191 exclusive 1 and itself, so judge if `a` has a divisor of 7 or 191, and note that 7 and 191 are prime numbers, `phi` of them is itself - 1, then we can use the Euler's theorem, see it on wiki https://en.wikipedia.org/wiki/Euler's_theorem, it's just Fermat's little theorem if the mod `n` is prime.

see how 1140 is calculated out:
phi(1337) = phi(7) * phi(191) = 6 * 190 = 1440

optimized solution update at 2016-7-12

Today, seeing @myanonymos 's comments, and I find several days ago I AC it just by fortunate coincidence, it's not the best solution. now I get a better idea.

(1) Firstly, if `a` has both divisor 7 and 191, that's `a % 1337 == 0`, answer is 0.
(2) Then if `a` has neither divisor 7 nor 191, that's a and 1337 are coprime, so $\mathbf{a^b \% 1337 = a^{b \% phi(1337)} \% 1337 = a^{b \% 1140} \% 1337}$.

(3) Finally, `a` could has either divisor 7 or 191, that's similar.
Let it be 7 for example.
Let $\mathbf{a = 7^n x}$
and let $\mathbf{b = 1140p + q}$, where $\mathbf{0 < q <= 1140}$
then:

$$\mathbf{a^b \% 1337}$$
$$\mathbf{= ((7^n x)^b) \% 1337}$$
$$\mathbf{= (7^{nb} x^b) \% 1337}$$
$$\mathbf{= ((7^{nb} \% 1337) * (x^b \% 1337)) \% 1337}$$
$$\mathbf{= ((7^{1140np + nq} \% 1337) * (x^{1140p + q} \% 1337)) \% 1337}$$

now note x and 1337 are coprime, so

$$\mathbf{= ((7^{1140np + nq} \% 1337) * (x^q \% 1337)) \% 1337}$$
$$\mathbf{= (7 * (7^{1140np + nq - 1} \% 191) * (x^q \% 1337)) \% 1337}$$

note 7 and 191 are coprime too, and 1140 is a multiple of 190, where 190 = phi(191). What's more we should assure that q != 0, if b % 1140== 0, then let b = 1140. so

$$\mathbf{= (7 * (7^{nq - 1} \% 191) * (x^q \% 1337)) \% 1337}$$
$$\mathbf{= ((7^{nq} \% 1337) * (x^q \% 1337)) \% 1337}$$
$$\mathbf{= (7^{nq} x^q) \% 1337}$$
$$\mathbf{= ((7^n x)^q) \% 1337}$$
$$\mathbf{= (a^q) \% 1337}$$

now you see condition (2) and (3) can be merged as one solution, if you take care of when `b % 1440 == 0`, and let `b += 1140`. Actually (1) can be merged too, but not efficient.

new code:
C++:

```
int superPow(int a, vector<int>& b) {
    if (a % 1337 == 0) return 0; // this line could also be removed
    int p = 0;
    for (int i : b) p = (p * 10 + i) % 1140;
    if (p == 0) p += 1140;
    return power(a, p, 1337);
}
int power(int x, int n, int mod) {
    int ret = 1;
    for (x %= mod; n; x = x * x % mod, n >>= 1) if (n & 1) ret = ret * x % mod;
    return ret;
}
```

java:

```
public int superPow(int a, int[] b) {
    if (a % 1337 == 0) return 0;
    int p = 0;
    for (int i : b) p = (p * 10 + i) % 1140;
    if (p == 0) p += 1440;
    return power(a, p, 1337);
}
public int power(int a, int n, int mod) {
    a %= mod;
    int ret = 1;
    while (n != 0) {
        if ((n & 1) != 0) ret = ret * a % mod;
        a = a * a % mod;
        n >>= 1;
    }
    return ret;
}
```

Actually, if p == 0 or not, we can always let p += 1140, it doesn't matter.
one line python:

```
def superPow(self, a, b):
    return 0 if a % 1337 == 0 else pow(a, reduce(lambda x, y: (x * 10 + y) % 1140
, b) + 1440, 1337)
```

will this be the best solution?

p.s.
I have testcases that the system missed
574
[1,1,4,0]

764
[1,1,4,0]
in this case if I remove this line of code `if (p == 0) p += 1140;`, it will goes wrong, but also can get AC on OJ.

and I found thar 574 * 574 % 1337 = 574, 764 * 764 % 1337 = 764, how interesting!

written by ShuangquanLi original link here

Solution 3

Solution 1: Using Python's big integers (accepted in 72 ms)

Turn `b` into a Python integer object (they grow arbitrarily large) and just use the `pow` function (which supports a modulo paramenter).

```
def superPow(self, a, b):
    return pow(a, int(''.join(map(str, b))), 1337)
```

Solution 2: Using small ints (accepted in 80 ms)

Originally I went backwards (see solution 5) but then I saw other people go forwards and it's simpler. Sigh. Anyway... my version:

```
def superPow(self, a, b):
    result = 1
    for digit in b:
        result = pow(result, 10, 1337) * pow(a, digit, 1337) % 1337
    return result
```

Explanation: For example for a^{5347}, the above computes a^5, then a^{53}, then a^{534}, and then finally a^{5347}. And a step from one to the next can be done like $a^{5347} = (a^{534})^{10} * a^7$.

Solution 3: Using recursion (accepted in 92 ms)

Obligatory recursive oneliner version of solution 2.

```
def superPow(self, a, b):
    return pow(a, b.pop(), 1337) * pow(self.superPow(a, b), 10, 1337) % 1337 if b
else 1
```

Solution 4: Using `reduce` (accepted in 80 ms)

Obligatory `reduce`-oneliner version of solution 2.

```
def superPow(self, a, b):
    return reduce(lambda result, digit: pow(result, 10, 1337) * pow(a, digit, 133
7) % 1337, b, 1)
```

Solution 5: omg was i stupid (accepted in 72 ms)

My original do-it-yourself before I saw other people's solutions and wrote solutions 2-4.

Using only small ints, also accepted in 72 ms:

```
def superPow(self, a, b):
    result = 1
    apower = a
    for digit in reversed(b):
        result = result * pow(apower, digit, 1337) % 1337
        apower = pow(apower, 10, 1337)
    return result
```

Explanation by example:

$a^{\mathbf{5347}}$
$= a^{5000} * a^{300} * a^{40} * a^{7}$
$= (a^{1000})^{5} * (a^{100})^{3} * (a^{10})^{4} * a^{7}$
$= (((a^{10})^{10})^{10})^{\mathbf{5}} * ((a^{10})^{10})^{\mathbf{3}} * (a^{10})^{\mathbf{4}} * a^{7}$

Computing that from back to front is straightforward (or straightbackward?).

written by StefanPochmann original link here

From Leetcoder.

Find K Pairs with Smallest Sums

You are given two integer arrays **nums1** and **nums2** sorted in ascending order and an integer **k**.

Define a pair **(u,v)** which consists of one element from the first array and one element from the second array.

Find the k pairs $(u_1,v_1),(u_2,v_2) \ldots (u_k,v_k)$ with the smallest sums.

Example 1:

```
Given nums1 = [1,7,11], nums2 = [2,4,6],  k = 3

Return: [1,2],[1,4],[1,6]

The first 3 pairs are returned from the sequence:
[1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
```

Example 2:

```
Given nums1 = [1,1,2], nums2 = [1,2,3],  k = 2

Return: [1,1],[1,1]

The first 2 pairs are returned from the sequence:
[1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]
```

Example 3:

```
Given nums1 = [1,2], nums2 = [3],  k = 3

Return: [1,3],[2,3]

All possible pairs are returned from the sequence:
[1,3],[2,3]
```

Credits:

Special thanks to @elmirap and @StefanPochmann for adding this problem and creating all test cases.

Solution 1

Several solutions from naive to more elaborate. I found it helpful to visualize the input as an **m×n matrix** of sums, for example for nums1=[1,7,11], and nums2=[2,4,6]:

```
      2   4   6
   +-----------
 1 |  3   5   7
 7 |  9  11  13
11 | 13  15  17
```

Of course the smallest pair overall is in the top left corner, the one with sum 3. We don't even need to look anywhere else. After including that pair in the output, the next-smaller pair must be the next on the right (sum=5) or the next below (sum=9). We can keep a "horizon" of possible candidates, implemented as a heap / priority-queue, and roughly speaking we'll grow from the top left corner towards the right/bottom. That's what my solution 5 does. Solution 4 is similar, not quite as efficient but a lot shorter and my favorite.

Solution 1: Brute Force (accepted in 560 ms)

Just produce all pairs, sort them by sum, and return the first k.

```
def kSmallestPairs(self, nums1, nums2, k):
    return sorted(itertools.product(nums1, nums2), key=sum)[:k]
```

Solution 2: Clean Brute Force (accepted in 532 ms)

The above produces tuples and while the judge doesn't care, it's cleaner to make them lists as requested:

```
def kSmallestPairs(self, nums1, nums2, k):
    return map(list, sorted(itertools.product(nums1, nums2), key=sum)[:k])
```

Solution 3: Less Brute Force (accepted in 296 ms)

Still going through all pairs, but only with a generator and `heapq.nsmallest`, which uses a heap of size k. So this only takes O(k) extra memory and O(mn log k) time.

```
def kSmallestPairs(self, nums1, nums2, k):
    return map(list, heapq.nsmallest(k, itertools.product(nums1, nums2), key=sum)
)
```

Or (accepted in 368 ms):

```
def kSmallestPairs(self, nums1, nums2, k):
    return heapq.nsmallest(k, ([u, v] for u in nums1 for v in nums2), key=sum)
```

Solution 4: Efficient (accepted in 112 ms)

The brute force solutions computed the whole matrix (see visualization above). This solution doesn't. It turns each row into a generator of triples [u+v, u, v], only computing the next when asked for one. And then merges these generators with a heap. Takes O(m + k*log(m)) time and O(m) extra space.

```
def kSmallestPairs(self, nums1, nums2, k):
    streams = map(lambda u: ([u+v, u, v] for v in nums2), nums1)
    stream = heapq.merge(*streams)
    return [suv[1:] for suv in itertools.islice(stream, k)]
```

Solution 5: More efficient (accepted in 104 ms)

The previous solution right away considered (the first pair of) all matrix rows (see visualization above). This one doesn't. It starts off only with the very first pair at the top-left corner of the matrix, and expands from there as needed. Whenever a pair is chosen into the output result, the next pair in the row gets added to the priority queue of current options. Also, if the chosen pair is the first one in its row, then the first pair in the next row is added to the queue.

```
def kSmallestPairs(self, nums1, nums2, k):
    queue = []
    def push(i, j):
        if i < len(nums1) and j < len(nums2):
            heapq.heappush(queue, [nums1[i] + nums2[j], i, j])
    push(0, 0)
    pairs = []
    while queue and len(pairs) < k:
        _, i, j = heapq.heappop(queue)
        pairs.append([nums1[i], nums2[j]])
        push(i, j + 1)
        if j == 0:
            push(i + 1, 0)
    return pairs
```

written by StefanPochmann original link here

Solution 2

Because both array are sorted, so we can keep track of the paired index. Therefore, we do not need to go through all combinations when k < nums1.length + num2.length. Time complexity is O(k*m) where m is the length of the shorter array.

```
public List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        List<int[]> ret = new ArrayList<int[]>();
        if (nums1.length == 0 || nums2.length == 0 || k == 0) {
            return ret;
        }

        int[] index = new int[nums1.length];
        while (k-- > 0) {
            int min_val = Integer.MAX_VALUE;
            int in = -1;
            for (int i = 0; i < nums1.length; i++) {
                if (index[i] >= nums2.length) {
                    continue;
                }
                if (nums1[i] + nums2[index[i]] < min_val) {
                    min_val = nums1[i] + nums2[index[i]];
                    in = i;
                }
            }
            if (in == -1) {
                break;
            }
            int[] temp = {nums1[in], nums2[index[in]]};
            ret.add(temp);
            index[in]++;
        }
        return ret;
    }
```

written by mylzsd original link here

Solution 3

Frist, we take the first k elements of nums1 and paired with nums2[0] as the starting pairs so that we have (0,0), (1,0), (2,0),.....(k-1,0) in the heap.
Each time after we pick the pair with min sum, we put the new pair with the second index +1. ie, pick (0,0), we put back (0,1). Therefore, the heap alway maintains at most min(k, len(nums1)) elements.

```
public class Solution {
    class Pair{
        int[] pair;
        int idx; // current index to nums2
        long sum;
        Pair(int idx, int n1, int n2){
            this.idx = idx;
            pair = new int[]{n1, n2};
            sum = (long) n1 + (long) n2;
        }
    }
    class CompPair implements Comparator<Pair> {
        public int compare(Pair p1, Pair p2){
            return Long.compare(p1.sum, p2.sum);
        }
    }
    public List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        List<int[]> ret = new ArrayList<>();
        if (nums1==null || nums2==null || nums1.length ==0 || nums2.length ==0) r
eturn ret;
        int len1 = nums1.length, len2=nums2.length;

        PriorityQueue<Pair> q = new PriorityQueue(k, new CompPair());
        for (int i=0; i<nums1.length && i<k ; i++) { // only need first k number
in nums1 to start
            q.offer( new Pair(0, nums1[i],nums2[0]) );
        }
        for (int i=1; i<=k && !q.isEmpty(); i++) { // get the first k sums
            Pair p = q.poll();
            ret.add( p.pair );
            if (p.idx < len2 -1 ) { // get to next value in nums2
                int next = p.idx+1;
                q.offer( new Pair(next, p.pair[0], nums2[next]) );
            }
        }
        return ret;
    }
}
```

written by yubad2000 original link here

Guess Number Higher or Lower

We are playing the Guess Game. The game is as follows:

I pick a number from **1** to ***n***. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number is higher or lower.

You call a pre-defined API `guess(int num)` which returns 3 possible results (`-1`, `1`, or `0`):

```
-1 : My number is lower
 1 : My number is higher
 0 : Congrats! You got it!
```

Example:

```
n = 10, I pick 6.

Return 6.
```

Solution 1

-1 : My number is lower
1 : My number is higher
0 : Congrats! You got it!

Here "My" means the number which is given for you to guess not the number you put into **guess(int num).**

written by Nakanu original link here

Solution 2

Using binary search to find the smallest number that's not too small.

```
    def guessNumber(self, n):
        class C: __getitem__ = lambda _, i: -guess(i)
        return bisect.bisect(C(), -1, 1, n)
```

Alternatively, without using the library:

```
    def guessNumber(self, n):
        lo, hi = 1, n
        while lo < hi:
            mid = (lo + hi) / 2
            if guess(mid) == 1:
                lo = mid + 1
            else:
                hi = mid
        return lo
```

Funny variation:

```
    def guessNumber(self, n):
        lo, hi = 1, n
        while lo < hi:
            mid = (lo + hi) / 2
            lo, hi = ((mid, mid), (mid+1, hi), (lo, mid-1))[guess(mid)]
        return lo
```

written by StefanPochmann original link here

Solution 3

This problem is a binary search problem has an O(logn) complexity.

```
public int guessNumber(int n) {
    int i = 1, j = n;
    while(i < j) {
        int mid = i + (j - i) / 2;
        if(guess(mid) == 0) {
            return mid;
        } else if(guess(mid) == 1) {
            i = mid + 1;
        } else {
            j = mid;
        }
    }
    return i;
}
```

written by yfcheng original link here

From Leetcoder.

Guess Number Higher or Lower II

We are playing the Guess Game. The game is as follows:

I pick a number from **1** to **n**. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number I picked is higher or lower.

However, when you guess a particular number x, and you guess wrong, you pay **$x**. You win the game when you guess the number I picked.

Example:

```
n = 10, I pick 8.

First round:  You guess 5, I tell you that it's higher. You pay $5.
Second round: You guess 7, I tell you that it's higher. You pay $7.
Third round:  You guess 9, I tell you that it's lower. You pay $9.

Game over. 8 is the number I picked.

You end up paying $5 + $7 + $9 = $21.
```

Given a particular **n ≥ 1**, find out how much money you need to have to guarantee a **win**.

1. The best strategy to play the game is to minimize the maximum loss you could possibly face. Another strategy is to minimize the expected loss. Here, we are interested in the **first** scenario.
2. Take a small example (n = 3). What do you end up paying in the worst case?
3. Check out this article if you're still stuck.
4. The purely recursive implementation of minimax would be worthless for even a small n. You MUST use dynamic programming.
5. As a follow-up, how would you modify your code to solve the problem of minimizing the expected loss, instead of the worst-case loss?

Credits:
Special thanks to @agave and @StefanPochmann for adding this problem and creating all test cases.

Solution 1

For each number x in range[i~j]
we do: result_when_pick_x = x + **max**{DP([i~x-1]), DP([x+1, j])}
--> *// the max means whenever you choose a number, the feedback is always bad and therefore leads you to a worse branch.*
then we get DP([i~j]) = **min**{xi, ... ,xj}
--> *// this min makes sure that you are minimizing your cost.*

```
public class Solution {
    public int getMoneyAmount(int n) {
        int[][] table = new int[n+1][n+1];
        return DP(table, 1, n);
    }

    int DP(int[][] t, int s, int e){
        if(s >= e) return 0;
        if(t[s][e] != 0) return t[s][e];
        int res = Integer.MAX_VALUE;
        for(int x=s; x<=e; x++){
            int tmp = x + Math.max(DP(t, s, x-1), DP(t, x+1, e));
            res = Math.min(res, tmp);
        }
        t[s][e] = res;
        return res;
    }
}
```

Here is a bottom up solution.

```
public class Solution {
    public int getMoneyAmount(int n) {
        int[][] table = new int[n+1][n+1];
        for(int j=2; j<=n; j++){
            for(int i=j-1; i>0; i--){
                int globalMin = Integer.MAX_VALUE;
                for(int k=i+1; k<j; k++){
                    int localMax = k + Math.max(table[i][k-1], table[k+1][j]);
                    globalMin = Math.min(globalMin, localMax);
                }
                table[i][j] = i+1==j?i:globalMin;
            }
        }
        return table[1][n];
    }
}
```

written by bbccyy1 original link here

Solution 2

Definition of `dp[i][j]`: minimum number of money to guarantee win for subproblem `[i, j]`.

Target: `dp[1][n]`

Corner case: `dp[i][i] = 0` (because the only element must be correct)

Equation: we can choose `k (i<=k<=j)` as our guess, and pay price `k`. After our guess, the problem is divided into two subproblems. Notice we do not need to pay the money for both subproblems. We only need to pay the worst case (because the system will tell us which side we should go) to guarantee win. So `dp[i][j] = min (i<=k<=j) { k + max(dp[i][k-1], dp[k+1][j]) }`

```
public class Solution {
    public int getMoneyAmount(int n) {
        if (n == 1) {
            return 0;
        }
        int[][] dp = new int[n + 1][n + 1];
        for (int jminusi = 1; jminusi < n; jminusi++) {
            for (int i = 0; i + jminusi <= n; i++) {
                int j = i + jminusi;
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = i; k <= j; k++) {
                    dp[i][j] = Math.min(dp[i][j],
                                        k + Math.max(k - 1 >= i ? dp[i][k - 1] :
0,
                                                     j >= k + 1 ? dp[k + 1][j] :
0));
                }
            }
        }
        return dp[1][n];
    }
}
```

written by dachuan.huang original link here

Solution 3

Big Idea: Given any n, we make a guess k. Then we break the interval [1,n] into [1,k - 1] and [k + 1,n]. The min of worst case cost can be calculated recursively as

cost[1,n] = k + max{cost[1,k - 1] + cost[k+1,n]}

Also, it takes a while for me to wrap my head around "min of max cost". My understand is that: you strategy is the best, but your luck is the worst. You only guess right when there is no possibilities to guess wrong.

```
public class Solution {
    public int getMoneyAmount(int n) {
        // all intervals are inclusive
        // uninitialized cells are assured to be zero
        // the zero column and row will be uninitialized
        // the illegal cells will also be uninitialized
        // add 1 to the length just to make the index the same as numbers used
        int[][] dp = new int[n + 1][n + 1]; // dp[i][j] means the min cost in the
worst case for numbers (i...j)

        // iterate the lengths of the intervals since the calculations of longer
intervals rely on shorter ones
        for (int l = 2; l <= n; l++) {
            // iterate all the intervals with length l, the start of which is i.
Hence the interval will be [i, i + (l - 1)]
            for (int i = 1; i <= n - (l - 1); i++) {
                dp[i][i + (l - 1)] = Integer.MAX_VALUE;
                // iterate all the first guesses g
                for (int g = i; g <= i + (l - 1); g++) {
                    int costForThisGuess;
                    // since if g is the last integer, g + 1 does not exist, we h
ave to separate this case
                    // cost for [i, i + (l - 1)]: g (first guess) + max{the cost
of left part [i, g - 1], the cost of right part [g + 1, i + (l - 1)]}
                    if (g == n) {
                        costForThisGuess = dp[i][g - 1] + g;
                    } else {
                        costForThisGuess = g + Math.max(dp[i][g - 1], dp[g + 1][
i + (l - 1)]);
                    }
                    dp[i][i + (l - 1)] = Math.min(dp[i][i + (l - 1)], costForThis
Guess); // keep track of the min cost among all first guesses
                }
            }
        }
        return dp[1][n];
    }
}
```

Any questions, suggestions & criticism welcomed!

written by myanonymos original link here

Wiggle Subsequence

A sequence of numbers is called a **wiggle sequence** if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a wiggle sequence.

For example, `[1,7,4,9,2,5]` is a wiggle sequence because the differences (6,-3,5,-7,3) are alternately positive and negative. In contrast, `[1,4,7,2,5]` and `[1,7,4,5,5]` are not wiggle sequences, the first because its first two differences are positive and the second because its last difference is zero.

Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original order.

Examples:

```
Input: [1,7,4,9,2,5]
Output: 6
The entire sequence is a wiggle sequence.

Input: [1,17,5,10,13,15,10,5,16,8]
Output: 7
There are several subsequences that achieve this length. One is [1,17,10,13,10,16,8
].

Input: [1,2,3,4,5,6,7,8,9]
Output: 2
```

Follow up:
Can you do it in O(n) time?

Credits:
Special thanks to @agave and @StefanPochmann for adding this problem and creating all test cases.

Solution 1

In Wiggle Subsequence, think that the solution we need should be in a way that we get alternative higher, lower,higher number.
Eg: 2, 5, 3, 8, 6, 9
In above example, the sequence of numbers is small,big,small,big,small,big numbers (In shape of hill).

Now for explanation, we take example series:
2,1,4,5,6,3,3,4,8,4

First we check if the series is starting as (big, small) or (small, big). So as 2,1 is big, small. So we will start the loop as we need small number first that is 1 as 2 is already there.

```
Step 1: First we check our requirement is to get small number. As 1<2 so the seri
es will be
 2,1
```

```
Step 2: Now we need big number that is  greater than 1. As 4>1 so series  will be
2,1,4
```

```
Step 3: Now we need small number. But 5>4 so 4 will be replaced by 5. So the seri
es will become
2,1,5
```

```
Step 4:  We need small number. But 6>5. Series will be
2,1,6
```

```
Step 5: Require small number. 3<6. Series will be
2,1,6,3
```

```
Step 6: Require big number. 3=3. No change in series
2,1,6,3
```

```
Step 7: Require big number. 4>3. Series will become
2,1,6,3,4
```

```
Step 8:  Require small number. 8>4. 8 will  replace 4 and series will become
2,1,6,3,8
```

```
Step 9: Require small number. 4<8. So final series will  be
2,1,6,3,8,4
```

Answer is 6.

In the code, for constant space O(1) we will modify the same 'num' array to store the (small, big, small) hill shape values. So the code will not only calculate the length of the sequence but if the interviewers asks for the Wiggle series also then we can return the series too. The leetcode Online Judge skipped a test case if the series starts with same set of numbers. Thanks to @ztq63830398, modified the code to consider that test case also.

Code:

```
    public class Solution {
 public int wiggleMaxLength(int[] nums) {
  if (nums.length == 0 || nums.length == 1) {
   return nums.length;
  }
  int k = 0;
  while (k < nums.length - 1 && nums[k] == nums[k + 1]) {  //Skips all the same n
umbers from series beginning eg 5, 5, 5, 1
   k++;
  }
  if (k == nums.length - 1) {
   return 1;
  }
  int result = 2;     // This will track the result of result array
  boolean smallReq = nums[k] < nums[k + 1];       //To check series starting patt
ern
  for (int i = k + 1; i < nums.length - 1; i++) {
   if (smallReq && nums[i + 1] < nums[i]) {
    nums[result] = nums[i + 1];
    result++;
    smallReq = !smallReq;    //Toggle the requirement from small to big number
   } else {
    if (!smallReq && nums[i + 1] > nums[i]) {
     nums[result] = nums[i + 1];
     result++;
     smallReq = !smallReq;    //Toggle the requirement from big to small number
    }
   }
  }
  return result;
 }
}
```

written by keshavk original link here

Solution 2

DP:

```
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
        int size=nums.size();
        if(size==0) return 0;
        vector<int> f(size, 1);
        vector<int> d(size, 1);
        for(int i=1; i<size; ++i){
            for(int j=0; j<i; ++j){
                if(nums[j]<nums[i]){
                    f[i]=max(f[i], d[j]+1);
                }
                else if(nums[j]>nums[i]){
                    d[i]=max(d[i], f[j]+1);
                }
            }
        }
        return max(d.back(), f.back());
    }
};
```

Greedy:

```
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
        int size=nums.size(), f=1, d=1;
        for(int i=1; i<size; ++i){
                 if(nums[i]>nums[i-1]) f=d+1;
            else if(nums[i]<nums[i-1]) d=f+1;
        }
        return min(size, max(f, d));
    }
};
```

written by clubmaster original link here

Solution 3

```
def wiggleMaxLength(self, nums):
    nan = float('nan')
    diffs = [a-b for a, b in zip([nan] + nums, nums + [nan]) if a-b]
    return sum(not d*e >= 0 for d, e in zip(diffs, diffs[1:]))
```

Explanation / Proof:

Imagine the given array contains [..., **10, 10, 10, 10**, ...]. Obviously we can't use more than one of those tens, as that wouldn't be wiggly. So right away we can ignore all consecutive duplicates.

Imagine the given array contains [..., 10, **7, 11, 13, 17, 19, 23** , 20, ...]. So increasing from 7 to 23. What can we do with that? Well we can't use more than two of those increasing numbers, as that wouldn't be wiggly. And if we do use two, we'd better use the 7 and the 23, as that offers the best extensibility (for example, the 19 wouldn't allow to next pick the 20 for the wiggly subsequence). And if we do use only one, it still should be either the 7 or the 23, as the 7 is the best wiggle-low and the 23 is the best wiggle-high of them. So whether we actually use the 7 and the 23 or not, we definitely can and should remove the 11, 13, 17, and 19. So then we have [..., 10, **7, 23**, 20, ...]. Now, notice that the 7 is a local minimum (both the 10 and the 23 are larger) and the 23 is a local maximum. And if we do this with **all** increasing or decreasing streaks, i.e., keep only their first and last number, then all the numbers we have left are local extrema, either smaller than both neighbors or larger than both neighbors. Which means that at that point, we're already fully wiggly. And we only removed as many numbers as we have to. So it's a longest possible wiggly subsequence.

My solution first computes differences of neighbors and throws out zeros (which does get rid of those useless consecutive duplicates). And then it just counts the local extrema (by checking two consecutive differences).

I use `nan` for some convenience, I'll let you figure that part out :-)

Alternative implementations not using the `nan` trick: First remove repetitions, then count the local extrema except the ends, and add the number of ends (because the ends are always local extrema)

```
def wiggleMaxLength(self, nums):
    norep = [num for num, _ in itertools.groupby(nums)]
    triples = zip(norep, norep[1:], norep[2:])
    return sum((b>a) == (b>c) for a, b, c in triples) + len(norep[:2])

def wiggleMaxLength(self, nums):
    norep = [num for num, _ in itertools.groupby(nums)]
    if len(norep) < 2: return len(norep)
    triples = zip(norep, norep[1:], norep[2:])
    return 2 + sum(a<b>c or a>b<c for a, b, c in triples)
```

written by StefanPochmann original link here

From Leetcoder.

Combination Sum IV

Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.

Example:

```
nums = [1, 2, 3]
target = 4

The possible combination ways are:
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)

Note that different sequences are counted as different combinations.

Therefore the output is 7.
```

Follow up:
What if negative numbers are allowed in the given array?
How does it change the problem?
What limitation we need to add to the question to allow negative numbers?

Credits:
Special thanks to @pbrother for adding this problem and creating all test cases.

Solution 1

Wish to learn better solutions from you guys.

```
public class Solution {
    public int combinationSum4(int[] nums, int target) {
        Arrays.sort(nums);
        int[] res = new int[target + 1];
        for (int i = 1; i < res.length; i++) {
     for (int num : nums) {
         if (num > i)
      break;
  else if (num == i)
      res[i] += 1;
  else
      res[i] += res[i-num];
     }
 }
        return res[target];
    }
}
```

written by feiyoshang original link here

Solution 2

Think about the recurrence relation first. How does the # of combinations of the `target` related to the # of combinations of numbers that are smaller than the `target`?

So we know that `target` is the sum of numbers in the array. Imagine we only need one more number to reach target, this number can be any one in the array, right? So the # of combinations of `target`, `comb[target] = sum(comb[target - nums[i]]), where 0 <= i < nums.length, and target >= nums[i]`.

In the example given, we can actually find the # of combinations of 4 with the # of combinations of 3(4 - 1), 2(4- 2) and 1(4 - 3). As a result, `comb[4] = comb[4-1] + comb[4-2] + comb[4-3] = comb[3] + comb[2] + comb[1]`.

Then think about the base case. Since if the target is 0, there is only one way to get zero, which is using 0, we can set `comb[0] = 1`.

Now we can come up with at least a recursive solution.

```
public int combinationSum4() {
    if (target == 0) {
        return 1;
    }
    int res = 0;
    for (int i = 0; i < nums.length; i++) {
        if (target >= nums[i]) {
            res += combinationSum4(nums, target - nums[i]);
        }
    }
    return res;
}
```

Now for a DP solution, we just need to figure out a way to store the intermediate results, to avoid the same combination sum being calculated many times. We can use an array to save those results, and check if there is already a result before calculation. We can fill the array with -1 to indicate that the result hasn't been calculated yet. 0 is not a good choice because it means there is no combination sum for the target.

```java
private int[] dp;

public int combinationSum4(int[] nums, int target) {
    dp = new int[target + 1];
    Arrays.fill(dp, -1);
    dp[0] = 1;
    return helper(nums, target);
}

private int helper(int[] nums, int target) {
    if (dp[target] != -1) {
        return dp[target];
    }
    int res = 0;
    for (int i = 0; i < nums.length; i++) {
        if (target >= nums[i]) {
            res += helper(nums, target - nums[i]);
        }
    }
    dp[target] = res;
    return res;
}
```

EDIT: The above solution is top-down. How about a bottom-up one?

```java
public int combinationSum4(int[] nums, int target) {
    int[] comb = new int[target + 1];
    comb[0] = 1;
    for (int i = 1; i < comb.length; i++) {
        for (int j = 0; j < nums.length; j++) {
            if (i - nums[j] >= 0) {
                comb[i] += comb[i - nums[j]];
            }
        }
    }
    return comb[target];
}
```

written by FreeTymeKiyan original link here

Solution 3

The DP solution goes through every possible sum from 1 to target one by one. Using recursion can skip those sums that are not the combinations of the numbers in the given array. Also, there is no need to sort the array first.

```
public class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    public int combinationSum4(int[] nums, int target) {
        int count = 0;
        if (nums == null || nums.length ==0 || target < 0 ) return 0;
        if ( target ==0 ) return 1;
        if (map.containsKey(target)) return map.get(target);
        for (int num: nums){
            count += combinationSum4(nums, target-num);
        }
        map.put(target, count);
        return count;
    }
}
```

written by yubad2000 original link here

From Leetcoder.

Kth Smallest Element in a Sorted Matrix

Given a *n* x *n* matrix where each of the rows and columns are sorted in ascending order, find the kth smallest element in the matrix.

Note that it is the kth smallest element in the sorted order, not the kth distinct element.

Example:

```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]
],
k = 8,

return 13.
```

Note:
You may assume k is always valid, $1 \le k \le n^2$.

Solution 1

Here is the step of my solution:

1. Build a minHeap of elements from the first row.
2. Do the following operations k-1 times :
   Every time when you poll out the root(Top Element in Heap), you need to know the row number and column number of that element(so we can create a tuple class here), replace that root with the next element from the same column.

After you finish this problem, thinks more :

1. For this question, you can also build a min Heap from the first column, and do the similar operations as above.(Replace the root with the next element from the same row)
2. What is more, this problem is exact the same with Leetcode373 Find K Pairs with Smallest Sums, I use the same code which beats 96.42%, after you solve this problem, you can check with this link: https://discuss.leetcode.com/topic/52953/share-my-solution-which-beat-96-42

```
public class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        int n = matrix.length;
        PriorityQueue<Tuple> pq = new PriorityQueue<Tuple>();
        for(int j = 0; j <= n-1; j++) pq.offer(new Tuple(0, j, matrix[0][j]));
        for(int i = 0; i < k-1; i++) {
            Tuple t = pq.poll();
            if(t.x == n-1) continue;
            pq.offer(new Tuple(t.x+1, t.y, matrix[t.x+1][t.y]));
        }
        return pq.poll().val;
    }
}

class Tuple implements Comparable<Tuple> {
    int x, y, val;
    public Tuple (int x, int y, int val) {
        this.x = x;
        this.y = y;
        this.val = val;
    }

    @Override
    public int compareTo (Tuple that) {
        return this.val - that.val;
    }
}
```

written by YUANGAO1023 original link here

Solution 2

```
class Solution
{
public:
 int kthSmallest(vector<vector<int>>& matrix, int k)
 {
  int n = matrix.size();
  int le = matrix[0][0], ri = matrix[n - 1][n - 1];
  int mid = 0;
  while (le < ri)
  {
   mid = (le + ri) >> 1;
   int num = 0;
   for (int i = 0; i < n; i++)
   {
    int pos = upper_bound(matrix[i].begin(), matrix[i].end(), mid) - matrix[i].be
gin();
    num += pos;
   }
   if (num < k)
   {
    le = mid + 1;
   }
   else
   {
    ri = mid;
   }
  }
  return le;
 }
};
```

written by 2997ms original link here

Solution 3

It's O(n) where n is the number of rows (and columns), not the number of elements. So it's very efficient. The algorithm is from the paper Selection in X + Y and matrices with sorted rows and columns, which I first saw mentioned by @elmirap (thanks).

The basic idea: Consider the submatrix you get by removing every second row and every second column. This has about a quarter of the elements of the original matrix. And the k-th element (k-th *smallest* I mean) of the original matrix is roughly the (k/4)-th element of the submatrix. So roughly get the (k/4)-th element of the submatrix and then use that to find the k-th element of the original matrix in O(n) time. It's recursive, going down to smaller and smaller submatrices until a trivial 2×2 matrix. For more details I suggest checking out the paper, the first half is easy to read and explains things well. Or @zhiqing_xiao's solution+explanation.

Cool: It uses variants of saddleback search that you might know for example from the Search a 2D Matrix II problem. And it uses the median of medians algorithm for linear-time selection.

Optimization: If k is less than n, we only need to consider the top-left k×k matrix. Similar if k is almost n^2. So it's even O(min(n, k, n^2-k)), I just didn't mention that in the title because I wanted to keep it simple and because those few very small or very large k are unlikely, most of the time k will be "medium" (and average $n^2/2$).

Implementation: I implemented the submatrix by using an index list through which the actual matrix data gets accessed. If [0, 1, 2, ..., n-1] is the index list of the original matrix, then [0, 2, 4, ...] is the index list of the submatrix and [0, 4, 8, ...] is the index list of the subsubmatrix and so on. This also covers the above optimization by starting with [0, 1, 2, ..., **k**-1] when applicable.

Application: I believe it can be used to easily solve the Find K Pairs with Smallest Sums problem in time O(k) instead of O(k log n), which I think is the best posted so far. I might try that later if nobody beats me to it (if you do, let me know :-).
Update: I did that now.

```
class Solution(object):
    def kthSmallest(self, matrix, k):

        # The median-of-medians selection function.
        def pick(a, k):
            if k == 1:
                return min(a)
            groups = (a[i:i+5] for i in range(0, len(a), 5))
            medians = [sorted(group)[len(group) / 2] for group in groups]
            pivot = pick(medians, len(medians) / 2 + 1)
            smaller = [x for x in a if x < pivot]
            if k <= len(smaller):
                return pick(smaller, k)
            k -= len(smaller) + a.count(pivot)
            return pivot if k < 1 else pick([x for x in a if x > pivot], k)

        # Find the k1-th and k2th smallest entries in the submatrix.
        def biselect(index, k1, k2):
```

```
        # Provide the submatrix.
        n = len(index)
        def A(i, j):
            return matrix[index[i]][index[j]]

        # Base case.
        if n <= 2:
            nums = sorted(A(i, j) for i in range(n) for j in range(n))
            return nums[k1-1], nums[k2-1]

        # Solve the subproblem.
        index_ = index[::2] + index[n-1+n%2:]
        k1_ = (k1 + 2*n) / 4 + 1 if n % 2 else n + 1 + (k1 + 3) / 4
        k2_ = (k2 + 3) / 4
        a, b = biselect(index_, k1_, k2_)

        # Prepare ra_less, rb_more and L with saddleback search variants.
        ra_less = rb_more = 0
        L = []
        jb = n   # jb is the first where A(i, jb) is larger than b.
        ja = n   # ja is the first where A(i, ja) is larger than or equal to
a.
        for i in range(n):
            while jb and A(i, jb - 1) > b:
                jb -= 1
            while ja and A(i, ja - 1) >= a:
                ja -= 1
            ra_less += ja
            rb_more += n - jb
            L.extend(A(i, j) for j in range(jb, ja))

        # Compute and return x and y.
        x = a if ra_less <= k1 - 1 else \
            b if k1 + rb_more - n*n <= 0 else \
            pick(L, k1 + rb_more - n*n)
        y = a if ra_less <= k2 - 1 else \
            b if k2 + rb_more - n*n <= 0 else \
            pick(L, k2 + rb_more - n*n)
        return x, y

    # Set up and run the search.
    n = len(matrix)
    start = max(k - n*n + n-1, 0)
    k -= n*n - (n - start)**2
    return biselect(range(start, min(n, start+k)), k, k)[0]
```

written by StefanPochmann original link here

Design Phone Directory

Design a Phone Directory which supports the following operations:

1. `get` : Provide a number which is not assigned to anyone.
2. `check` : Check if a number is available or not.
3. `release` : Recycle or release a number.

Example:

```
// Init a phone directory containing a total of 3 numbers: 0, 1, and 2.
PhoneDirectory directory = new PhoneDirectory(3);

// It can return any available phone number. Here we assume it returns 0.
directory.get();

// Assume it returns 1.
directory.get();

// The number 2 is available, so return true.
directory.check(2);

// It returns 2, the only number that is left.
directory.get();

// The number 2 is no longer available, so return false.
directory.check(2);

// Release number 2 back to the pool.
directory.release(2);

// Number 2 is available again, return true.
directory.check(2);
```

Solution 1

```
    Set<Integer> used = new HashSet<Integer>();
    Queue<Integer> available = new LinkedList<Integer>();
    int max;
    public PhoneDirectory(int maxNumbers) {
        max = maxNumbers;
        for (int i = 0; i < maxNumbers; i++) {
            available.offer(i);
        }
    }

    public int get() {
        Integer ret = available.poll();
        if (ret == null) {
            return -1;
        }
        used.add(ret);
        return ret;
    }

    public boolean check(int number) {
        if (number >= max || number < 0) {
            return false;
        }
        return !used.contains(number);
    }

    public void release(int number) {
        if (used.remove(number)) {
            available.offer(number);
        }
    }
```

written by mylzsd original link here

Solution 2

The idea is to use java's bitset and use smallestFreeIndex/max to keep track of the limit.
Also, by keeping track of the updated smallestFreeIndex all the time, the run time of get()
is spared from scanning the entire bitset every time.

```
public class PhoneDirectory {

    BitSet bitset;
    int max; // max limit allowed
    int smallestFreeIndex; // current smallest index of the free bit

    public PhoneDirectory(int maxNumbers) {
        this.bitset = new BitSet(maxNumbers);
        this.max = maxNumbers;
    }

    public int get() {
        // handle bitset fully allocated
        if(smallestFreeIndex == max) {
            return -1;
        }
        int num = smallestFreeIndex;
        bitset.set(smallestFreeIndex);
        //Only scan for the next free bit, from the previously known smallest fre
e index
        smallestFreeIndex = bitset.nextClearBit(smallestFreeIndex);
        return num;
    }

    public boolean check(int number) {
        return bitset.get(number) == false;
    }

    public void release(int number) {
        //handle release of unallocated ones
        if(bitset.get(number) == false)
            return;
        bitset.clear(number);
        if(number < smallestFreeIndex) {
            smallestFreeIndex = number;
        }
    }
}
```

written by johnyrufus16 original link here

Solution 3

```
    def __init__(self, maxNumbers):
        self.available = set(range(maxNumbers))

    def get(self):
        return self.available.pop() if self.available else -1

    def check(self, number):
        return number in self.available

    def release(self, number):
        self.available.add(number)
```

written by agave original link here

From Leetcoder.

Insert Delete GetRandom O(1)

Design a data structure that supports all following operations in **O(1)** time.

1. `insert(val)` : Inserts an item val to the set if not already present.
2. `remove(val)` : Removes an item val from the set if present.
3. `getRandom` : Returns a random element from current set of elements. Each element must have the **same probability** of being returned.

Example:

```
// Init an empty set.
RandomizedSet randomSet = new RandomizedSet();

// Inserts 1 to the set. Returns true as 1 was inserted successfully.
randomSet.insert(1);

// Returns false as 2 does not exist in the set.
randomSet.remove(2);

// Inserts 2 to the set, returns true. Set now contains [1,2].
randomSet.insert(2);

// getRandom should return either 1 or 2 randomly.
randomSet.getRandom();

// Removes 1 from the set, returns true. Set now contains [2].
randomSet.remove(1);

// 2 was already in the set, so return false.
randomSet.insert(2);

// Since 1 is the only number in the set, getRandom always return 1.
randomSet.getRandom();
```

Solution 1

I got a similar question for my phone interview. The difference is that the duplicated number is allowed. So, think that is a follow-up of this question.
How do you modify your code to allow duplicated number?

```
public class RandomizedSet {
    ArrayList<Integer> nums;
    HashMap<Integer, Integer> locs;
    java.util.Random rand = new java.util.Random();
    /** Initialize your data structure here. */
    public RandomizedSet() {
        nums = new ArrayList<Integer>();
        locs = new HashMap<Integer, Integer>();
    }

    /** Inserts a value to the set. Returns true if the set did not already conta
in the specified element. */
    public boolean insert(int val) {
        boolean contain = locs.containsKey(val);
        if ( contain ) return false;
        locs.put( val, nums.size());
        nums.add(val);
        return true;
    }

    /** Removes a value from the set. Returns true if the set contained the speci
fied element. */
    public boolean remove(int val) {
        boolean contain = locs.containsKey(val);
        if ( ! contain ) return false;
        int loc = locs.get(val);
        if (loc < nums.size() - 1 ) { // not the last one than swap the last one
with this val
            int lastone = nums.get(nums.size() - 1 );
            nums.set( loc , lastone );
            locs.put(lastone, loc);
        }
        locs.remove(val);
        nums.remove(nums.size() - 1);
        return true;
    }

    /** Get a random element from the set. */
    public int getRandom() {
        return nums.get( rand.nextInt(nums.size()) );
    }
}
```

written by yubad2000 original link here

Solution 2

```
class RandomizedSet {
public:
    /** Initialize your data structure here. */
    RandomizedSet() {

    }

    /** Inserts a value to the set. Returns true if the set did not already conta
in the specified element. */
    bool insert(int val) {
        if (m.find(val) != m.end()) return false;
        nums.emplace_back(val);
        m[val] = nums.size() - 1;
        return true;
    }

    /** Removes a value from the set. Returns true if the set contained the speci
fied element. */
    bool remove(int val) {
        if (m.find(val) == m.end()) return false;
        int last = nums.back();
        m[last] = m[val];
        nums[m[val]] = last;
        nums.pop_back();
        m.erase(val);
        return true;
    }

    /** Get a random element from the set. */
    int getRandom() {
        return nums[rand() % nums.size()];
    }
private:
    vector<int> nums;
    unordered_map<int, int> m;
};
```

written by Ren.W original link here

Solution 3

```java
public class RandomizedSet {
    HashMap<Integer, Integer> map;
    ArrayList<Integer> list;

    /** Initialize your data structure here. */
    public RandomizedSet() {
        map = new HashMap<Integer, Integer>();
        list = new ArrayList<Integer>();
    }

    /** Inserts a value to the set. Returns true if the set did not already conta
in the specified element. */
    public boolean insert(int val) {
        if(map.containsKey(val)) {
            return false;
        }else {
            map.put(val, list.size());
            list.add(val);
            return true;
        }
    }

    /** Removes a value from the set. Returns true if the set contained the speci
fied element. */
    public boolean remove(int val) {
        if(!map.containsKey(val)) {
            return false;
        }else {
            int key = map.get(val);
            int lastElement = list.get(list.size() - 1);
            map.put(lastElement, key);
            list.set(key, lastElement);
            map.remove(val);
            list.remove(list.size() - 1);
            return true;
        }
    }

    /** Get a random element from the set. */
    public int getRandom() {
        Random random = new Random();
        return list.get( random.nextInt(list.size()) );
    }
}

/**
 * Your RandomizedSet object will be instantiated and called as such:
 * RandomizedSet obj = new RandomizedSet();
 * boolean param_1 = obj.insert(val);
 * boolean param_2 = obj.remove(val);
 * int param_3 = obj.getRandom();
 */
```

written by vivian34721 original link here

From Leetcoder.

Insert Delete GetRandom O(1) - Duplicates allowed

Design a data structure that supports all following operations in *average* **O(1)** time.

Note: Duplicate elements are allowed.

1. `insert(val)` : Inserts an item val to the collection.
2. `remove(val)` : Removes an item val from the collection if present.
3. `getRandom` : Returns a random element from current collection of elements. The probability of each element being returned is **linearly related** to the number of same value the collection contains.

Example:

```
// Init an empty collection.
RandomizedCollection collection = new RandomizedCollection();

// Inserts 1 to the collection. Returns true as the collection did not contain 1.
collection.insert(1);

// Inserts another 1 to the collection. Returns false as the collection contained 1
. Collection now contains [1,1].
collection.insert(1);

// Inserts 2 to the collection, returns true. Collection now contains [1,1,2].
collection.insert(2);

// getRandom should return 1 with the probability 2/3, and returns 2 with the proba
bility 1/3.
collection.getRandom();

// Removes 1 from the collection, returns true. Collection now contains [1,2].
collection.remove(1);

// getRandom should return 1 and 2 both equally likely.
collection.getRandom();
```

Solution 1

The problem is a simple extension of the previous problem that did not have duplicates. Instead of storing a single index like in the previous problem, we simply store a collection of indices for all the times that a number appears in the array.

Insert() and random() are quite straightforward. For remove(), we take advantage of the fact that adding/removing from a HashSet is O(1) average time. The logic is otherwise similar - swap the index of any one instance of the item to be removed with the item in the very last place of the array. Update the sets after doing so, and then remove the last item.

Thanks to @yubad2000 for the wonderful idea of using a LinkedHashSet for O(1) iteration over large items. An iterator over a normal HashSet is actually O(h/n), where h is table capacity. So it is not a solution to our problem requiring O(1) time. Nor does an ArrayList instead of a HashSet work (I wasted some time on that for a while...).

```
public class RandomizedCollection {

    ArrayList<Integer> result;
    HashMap<Integer, LinkedHashSet<Integer>> map;

    public RandomizedCollection() {
        result = new ArrayList<Integer>();
        map = new HashMap<Integer, LinkedHashSet<Integer>>();
    }

    /** Inserts a value to the collection. Returns true if the collection did not
already contain the specified element. */
    public boolean insert(int val) {
        // Add item to map if it doesn't already exist.
        boolean alreadyExists = map.containsKey(val);
        if(!alreadyExists) {
            map.put(val, new LinkedHashSet<Integer>());
        }
        map.get(val).add(result.size());
        result.add(val);
        return !alreadyExists;
    }

    /** Removes a value from the collection. Returns true if the collection conta
ined the specified element. */
    public boolean remove(int val) {
        if(!map.containsKey(val)) {
            return false;
        }
        // Get arbitary index of the ArrayList that contains val
        LinkedHashSet<Integer> valSet = map.get(val);
        int indexToReplace = valSet.iterator().next();

        // Obtain the set of the number in the last place of the ArrayList
        int numAtLastPlace = result.get(result.size() - 1);
        LinkedHashSet<Integer> replaceWith = map.get(numAtLastPlace);
```

```java
        // Replace val at arbitary index with very last number
        result.set(indexToReplace, numAtLastPlace);

        // Remove appropriate index
        valSet.remove(indexToReplace);

        // Don't change set if we were replacing the removed item with the same n
umber
        if(indexToReplace != result.size() - 1) {
            replaceWith.remove(result.size() - 1);
            replaceWith.add(indexToReplace);
        }
        result.remove(result.size() - 1);

        // Remove map entry if set is now empty, then return
        if(valSet.isEmpty()) {
            map.remove(val);
        }
        return true;
    }

    /** Get a random element from the collection. */
    public int getRandom() {
        // Get linearly random item
        return result.get((int)(Math.random() * result.size()));
    }
}
```

written by DeusVult original link here

Solution 2

This post is deleted!
written by haruhiku original link here

Solution 3

This solution assumes there is not ordering requirement for `remove`. `remove` will randomly remove an element among duplicates.

Similar to Problem 380 without duplicates, we use a list to contain all elements and use a dict to keep track of their indices. This time, we use a `set` to keep track of all indices for a value.

Just like Problem 380, when we remove a value, we randomly pop an index for that value from the set, swap the value with the last element of the list, and update the indices for the new value.

```
import random
class RandomizedCollection(object):

    def __init__(self):
        self.l, self.d = [], collections.defaultdict(set)

    def insert(self, val):
        self.d[val].add(len(self.l))
        self.l.append(val)
        return len(self.d[val])==1

    def remove(self, val):
        if val not in self.d:
            return False
        i, newVal = self.d[val].pop(), self.l[-1]
        len(self.d[val]) > 0 or self.d.pop(val, None)
        if newVal in self.d:
            self.d[newVal] = (self.d[newVal] | {i}) - {len(self.l)-1}
        self.l[i] = newVal
        self.l.pop()
        return True

    def getRandom(self):
        return random.choice(self.l)
```

Longer but clearer version:

```python
import random
class RandomizedCollection(object):
    def __init__(self):
        self.l = []
        self.d = collections.defaultdict(set)

    def insert(self, val):
        b = val not in self.d
        self.d[val].add(len(self.l))
        self.l.append(val)
        return b

    def remove(self, val):
        if val not in self.d:
            return False
        i, newVal = self.d[val].pop(), self.l[-1]
        if len(self.d[val]) == 0:
            del self.d[val]
        self.l[i] = newVal
        if newVal in self.d:
            self.d[newVal].add(i)
            self.d[newVal].discard(len(self.l)-1)
        self.l.pop()
        return True

    def getRandom(self):
        return random.choice(self.l)
```

written by o_sharp original link here

From Leetcoder.

Linked List Random Node

Given a singly linked list, return a random node's value from the linked list. Each node must have the **same probability** of being chosen.

Follow up:
What if the linked list is extremely large and its length is unknown to you? Could you solve this efficiently without using extra space?

Example:

```
// Init a singly linked list [1,2,3].
ListNode head = new ListNode(1);
head.next = new ListNode(2);
head.next.next = new ListNode(3);
Solution solution = new Solution(head);

// getRandom() should return either 1, 2, or 3 randomly. Each element should have e
qual probability of returning.
solution.getRandom();
```

Solution 1

Problem:

- Choose `k` entries from `n` numbers. Make sure each number is selected with the probability of `k/n`

Basic idea:

- Choose `1, 2, 3, ..., k` first and put them into the reservoir.
- For `k+1`, pick it with a probability of `k/(k+1)`, and randomly replace a number in the reservoir.
- For `k+i`, pick it with a probability of `k/(k+i)`, and randomly replace a number in the reservoir.
- Repeat until `k+i` reaches `n`

Proof:

- For `k+i`, the probability that it is selected and will replace a number in the reservoir is `k/(k+i)`
- For a number in the reservoir before (let's say `X`), the probability that it keeps staying in the reservoir is
  - `P(X was in the reservoir last time)` × `P(X is not replaced by k+i)`
  - = `P(X was in the reservoir last time)` × (`1` - `P(k+i is selected and replaces X)`)
  - = `k/(k+i-1)` × (`1` - `k/(k+i)` × `1/k`)
  - = `k/(k+i)`
- When `k+i` reaches `n`, the probability of each number staying in the reservoir is `k/n`

Example

- Choose `3` numbers from `[111, 222, 333, 444]`. Make sure each number is selected with a probability of `3/4`
- First, choose `[111, 222, 333]` as the initial reservior
- Then choose `444` with a probability of `3/4`
- For `111`, it stays with a probability of
  - `P(444 is not selected)` + `P(444 is selected but it replaces 222 or 333)`
  - = `1/4` + `3/4` * `2/3`
  - = `3/4`
- The same case with `222` and `333`
- Now all the numbers have the probability of `3/4` to be picked

This Problem <Linked List Random Node>

- This problem is the sp case where `k=1`

P.S. Thanks for @WKVictor for pointing out my mistake!

written by WTIFS original link here

Solution 2

```
import java.util.*;
public class Solution {

    /** @param head The linked list's head. Note that the head is guanranteed to
be not null, so it contains at least one node. */
    ListNode head = null;
    Random randomGenerator = null;
    public Solution(ListNode head) {
        this.head = head;
        this.randomGenerator = new Random();

    }

    /** Returns a random node's value. */
    public int getRandom() {
        ListNode result = null;
        ListNode current = head;

        for(int n = 1; current!=null; n++) {
            if (randomGenerator.nextInt(n) == 0) {
                result = current;
            }
            current = current.next;
        }

        return result.val;

    }
}

/**
 * Your Solution object will be instantiated and called as such:
 * Solution obj = new Solution(head);
 * int param_1 = obj.getRandom();
 */
```

written by abi93k original link here

Solution 3

according to the wiki https://en.wikipedia.org/wiki/Reservoir_sampling
here is sudo code for k size reservoir:

```
/*
  S has items to sample, R will contain the result
*/
ReservoirSample(S[1..n], R[1..k])
  // fill the reservoir array
  for i = 1 to k
      R[i] := S[i]

  // replace elements with gradually decreasing probability
  for i = k+1 to n
    j := random(1, i)   // important: inclusive range
    if j <= k
        R[j] := S[i]
```

you need to remember the range [0, i] should be inclusive.

```
class Solution {
private:
    ListNode* head;
public:
    /** @param head The linked list's head. Note that the head is guanranteed to
be not null, so it contains at least one node. */
    Solution(ListNode* head) {
        this->head = head;
    }

    /** Returns a random node's value. */
    int getRandom() {
        int res = head->val;
        ListNode* node = head->next;
        int i = 2;
        while(node){
            int j = rand()%i;
            if(j==0)
                res = node->val;
            i++;
            node = node->next;
        }
        return res;
    }
};
```

written by primbo original link here

Ransom Note

Given an arbitrary ransom note string and another string containing letters from all the magazines, write a function that will return true if the ransom note can be constructed from the magazines ; otherwise, it will return false.

Each letter in the magazine string can only be used once in your ransom note.

Note:
You may assume that both strings contain only lowercase letters.

```
canConstruct("a", "b") -> false
canConstruct("aa", "ab") -> false
canConstruct("aa", "aab") -> true
```

Solution 1

```java
public class Solution {
    public boolean canConstruct(String ransomNote, String magazine) {
        int[] arr = new int[26];
        for (int i = 0; i < magazine.length(); i++) {
            arr[magazine.charAt(i) - 'a']++;
        }
        for (int i = 0; i < ransomNote.length(); i++) {
            if(--arr[ransomNote.charAt(i)-'a'] < 0) {
                return false;
            }
        }
        return true;
    }
}
```

written by yidongwang original link here

Solution 2

O(m+n) with m and n being the lengths of the strings.

```
def canConstruct(self, ransomNote, magazine):
    return not collections.Counter(ransomNote) - collections.Counter(magazine)
```

written by StefanPochmann original link here

Solution 3

The complexity is O(N), N is the length of magazine.

```
class Solution {
public:
    bool canConstruct(string ransomNote, string magazine) {
        unordered_map<char, int> map;
        for (int i = 0; i < magazine.size(); ++i)
            ++map[magazine[i]];
        for (int j = 0; j < ransomNote.size(); ++j)
            if (--map[ransomNote[j]] < 0)
                return false;
        return true;
    }
};
```

Or you can use a vector with size 26 instead of an unordered_map.

```
class Solution {
public:
    bool canConstruct(string ransomNote, string magazine) {
        vector<int> vec(26, 0);
        for (int i = 0; i < magazine.size(); ++i)
            ++vec[magazine[i] - 'a'];
        for (int j = 0; j < ransomNote.size(); ++j)
            if (--vec[ransomNote[j] - 'a'] < 0)
                return false;
        return true;
    }
};
```

I remember that there are two variations of this question, perhaps they will come in the next few days :)

1. If you can only pick letters from the magazine in order.
2. If the magazine is double sided.

written by haruhiku original link here

From Leetcoder.

Shuffle an Array

Shuffle a set of numbers without duplicates.

Example:

```
// Init an array with set 1, 2, and 3.
int[] nums = {1,2,3};
Solution solution = new Solution(nums);

// Shuffle the array [1,2,3] and return its result. Any permutation of [1,2,3] must
 equally likely to be returned.
solution.shuffle();

// Resets the array back to its original configuration [1,2,3].
solution.reset();

// Returns the random shuffling of array [1,2,3].
solution.shuffle();
```

Solution 1

```java
import java.util.Random;

public class Solution {
    private int[] nums;
    private Random random;

    public Solution(int[] nums) {
        this.nums = nums;
        random = new Random();
    }

    /** Resets the array to its original configuration and return it. */
    public int[] reset() {
        return nums;
    }

    /** Returns a random shuffling of the array. */
    public int[] shuffle() {
        if(nums == null) return null;
        int[] a = nums.clone();
        for(int j = 1; j < a.length; j++) {
            int i = random.nextInt(j + 1);
            swap(a, i, j);
        }
        return a;
    }

    private void swap(int[] a, int i, int j) {
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
}
```

written by qianzhige original link here

Solution 2

Straight-forward solution

```
class Solution {
    vector<int> nums;
public:
    Solution(vector<int> nums) {
        this->nums = nums;
    }

    /** Resets the array to its original configuration and return it. */
    vector<int> reset() {
        return nums;
    }

    /** Returns a random shuffling of the array. */
    vector<int> shuffle() {
        vector<int> result(nums);
        for (int i = 0;i < result.size();i++) {
            int pos = rand()%(result.size()-i);
            swap(result[i+pos], result[i]);
        }
        return result;
    }
};

/**
 * Your Solution object will be instantiated and called as such:
 * Solution obj = new Solution(nums);
 * vector<int> param_1 = obj.reset();
 * vector<int> param_2 = obj.shuffle();
 */
```

written by Louis1992 original link here

Solution 3

Just for fun.

```
class Solution(object):
    def __init__(self, nums):
        self.reset = lambda: nums
        self.shuffle = lambda: random.sample(nums, len(nums))
```

written by StefanPochmann original link here

From Leetcoder.

Mini Parser

Given a nested list of integers represented as a string, implement a parser to deserialize it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Note: You may assume that the string is well-formed:

- String is non-empty.
- String does not contain white spaces.
- String contains only digits `0-9`, `[`, `-` `,`, `]`.

Example 1:

```
Given s = "324",

You should return a NestedInteger object which contains a single integer 324.
```

Example 2:

```
Given s = "[123,[456,[789]]]",

Return a NestedInteger object containing a nested list with 2 elements:

1. An integer containing value 123.
2. A nested list containing two elements:
    i.  An integer containing value 456.
    ii. A nested list with one element:
         a. An integer containing value 789.
```

Solution 1

This approach will just iterate through every char in the string (no recursion).

- If encounters '[', push current NestedInteger to stack and start a new one.
- If encounters ']', end current NestedInteger and pop a NestedInteger from stack to continue.
- If encounters ',', append a new number to curr NestedInteger, if this comma is not right after a brackets.
- Update index l and r, where l shall point to the start of a integer substring, while r shall points to the end+1 of substring.

Java Code:

```
public NestedInteger deserialize(String s) {
    if (s.isEmpty())
        return null;
    if (s.charAt(0) != '[') // ERROR: special case
        return new NestedInteger(Integer.valueOf(s));

    Stack<NestedInteger> stack = new Stack<>();
    NestedInteger curr = null;
    int l = 0; // l shall point to the start of a number substring;
               // r shall point to the end+1 of a number substring
    for (int r = 0; r < s.length(); r++) {
        char ch = s.charAt(r);
        if (ch == '[') {
            if (curr != null) {
                stack.push(curr);
            }
            curr = new NestedInteger();
            l = r+1;
        } else if (ch == ']') {
            String num = s.substring(l, r);
            if (!num.isEmpty())
                curr.add(new NestedInteger(Integer.valueOf(num)));
            if (!stack.isEmpty()) {
                NestedInteger pop = stack.pop();
                pop.add(curr);
                curr = pop;
            }
            l = r+1;
        } else if (ch == ',') {
            if (s.charAt(r-1) != ']') {
                String num = s.substring(l, r);
                curr.add(new NestedInteger(Integer.valueOf(num)));
            }
            l = r+1;
        }
    }

    return curr;
}
```

written by AlexTheGreat original link here

Solution 2

Python using `eval`:

```
def deserialize(self, s):
    def nestedInteger(x):
        if isinstance(x, int):
            return NestedInteger(x)
        lst = NestedInteger()
        for y in x:
            lst.add(nestedInteger(y))
        return lst
    return nestedInteger(eval(s))
```

Python one-liner

```
def deserialize(self, s):
    return NestedInteger(s) if isinstance(s, int) else reduce(lambda a, x: a.add(
self.deserialize(x)) or a, s, NestedInteger()) if isinstance(s, list) else self.d
eserialize(eval(s))
```

Python Golf (136 bytes or 31 bytes)

```
class Solution:deserialize=d=lambda S,s,N=NestedInteger:s<[]and N(s)or s<''and red
uce(lambda a,x:a.add(S.d(x))or a,s,N())or S.d(eval(s))
```

Or abusing how the judge judges (yes, this gets accepted):

```
class Solution:deserialize=eval
```

Python parsing char by char

Here I turned the input string into a list with sentinel for convenience.

```python
def deserialize(self, s):
    def nestedInteger():
        num = ''
        while s[-1] in '1234567890-':
            num += s.pop()
        if num:
            return NestedInteger(int(num))
        s.pop()
        lst = NestedInteger()
        while s[-1] != ']':
            lst.add(nestedInteger())
            if s[-1] == ',':
                s.pop()
        s.pop()
        return lst
    s = list(' ' + s[::-1])
    return nestedInteger()
```

C++ using `istringstream`

```cpp
class Solution {
public:
    NestedInteger deserialize(string s) {
        istringstream in(s);
        return deserialize(in);
    }
private:
    NestedInteger deserialize(istringstream &in) {
        int number;
        if (in >> number)
            return NestedInteger(number);
        in.clear();
        in.get();
        NestedInteger list;
        while (in.peek() != ']') {
            list.add(deserialize(in));
            if (in.peek() == ',')
                in.get();
        }
        in.get();
        return list;
    }
};
```

written by StefanPochmann original link here

Solution 3

Solution in a Glance:
This solution uses a stack to record the NestedInteger's.
At the very beginning, an empty NestedInteger is placed in the stack. This NestedInteger will be regarded as a list that holds one but only one NestedInteger, which will be returned in the end.
Logic: When encountering '[', the stack has one more element. When encountering ']', the stack has one less element.

Complexities:

- Time: $O(n)$
- Space: worse-case $O(n)$ (worse case: [1,[2,[3,[....[n-1,[n]]]....])

C++ Accepted Code:
(A possible implementation of **NestedInteger** is also provided in the first reply.)

```
class Solution {
public:
    NestedInteger deserialize(string s) {
        function<bool(char)> isnumber = [](char c){ return (c == '-') || isdigit(
c); };

        stack<NestedInteger> stk;
        stk.push(NestedInteger());

        for (auto it = s.begin(); it != s.end();) {
            const char & c = (*it);
            if (isnumber(c)) {
                auto it2 = find_if_not(it, s.end(), isnumber);
                int val = stoi(string(it, it2));
                stk.top().add(NestedInteger(val));
                it = it2;
            }
            else {
                if (c == '[') {
                    stk.push(NestedInteger());
                }
                else if (c == ']') {
                    NestedInteger ni = stk.top();
                    stk.pop();
                    stk.top().add(ni);
                }
                ++it;
            }
        }

        NestedInteger result = stk.top().getList().front();
        return result;
    }
};
```

written by zhiqing_xiao original link here

From Leetcoder.

Lexicographical Numbers

Given an integer *n*, return 1 - *n* in lexicographical order.

For example, given 13, return: [1,10,11,12,13,2,3,4,5,6,7,8,9].

Please optimize your algorithm to use less time and space. The input size may be as large as 5,000,000.

Solution 1

```
public List<Integer> lexicalOrder(int n) {
        List<Integer> list = new ArrayList<>(n);
        int curr = 1;
        for (int i = 1; i <= n; i++) {
            list.add(curr);
            if (curr * 10 <= n) {
                curr *= 10;
            } else if (curr % 10 != 9 && curr + 1 <= n) {
                curr++;
            } else {
                while ((curr / 10) % 10 == 9) {
                    curr /= 10;
                }
                curr = curr / 10 + 1;
            }
        }
        return list;
    }
```

The basic idea is to find the next number to add.
Take 45 for example: if the current number is 45, the next one will be 450 (450 == 45 * 10)(if 450 <= n), or 46 (46 == 45 + 1) (if 46 <= n) or 5 (5 == 45 / 10 + 1)(5 is less than 45 so it is for sure less than n).
We should also consider n = 600, and the current number = 499, the next number is 5 because there are all "9"s after "4" in "499" so we should divide 499 by 10 until the last digit is not "9".
It is like a tree, and we are easy to get a sibling, a left most child and the parent of any node.

written by songzec original link here

Solution 2

```
The idea is pretty simple. If we look at the order we can find out we just keep ad
ding digit from 0 to 9 to every digit and make it a tree.
Then we visit every node in pre-order.
       1        2        3    ...
      /\        /\       /\
   10 ...19  20...29  30...39   ....
```

```java
public class Solution {
    public List<Integer> lexicalOrder(int n) {
        List<Integer> res = new ArrayList<>();
        for(int i=1;i<10;++i){
          dfs(i, n, res);
        }
        return res;
    }

    public void dfs(int cur, int n, List<Integer> res){
        if(cur>n)
            return;
        else{
            res.add(cur);
            for(int i=0;i<10;++i){
                if(10*cur+i>n)
                    return;
                dfs(10*cur+i, n, res);
            }
        }
    }
}
```

written by xialanxuan1015 original link here

Solution 3

Three accepted solutions and me rambling on about failed attempts :-D

Solution 1 (accepted in 1792, 1747, 1700 ms)

I just sort the numbers 1 to n using my custom comparison. To compare two numbers, I "left-shift" them both before comparing them. For example if n = 49999, then I left-shift numbers so they're five digits. That is, 42 becomes 42000 and 123 becomes 12300. In case of ties, e.g., 420 also becoming 42000, the stability of `sorted` keeps them in order.

```
def lexicalOrder(self, n):
    top = 1
    while top * 10 <= n:
        top *= 10
    def mycmp(a, b, top=top):
        while a < top: a *= 10
        while b < top: b *= 10
        return -1 if a < b else b < a
    return sorted(xrange(1, n+1), mycmp)
```

Solution 2 (accepted in 1268, 1508, 1320, 1356, 1336 ms)

```
def lexicalOrder(self, n):
    withKeys = []
    for i in xrange(1, n+1):
        key = i
        while key < 1000000:
            key *= 10
        withKeys.append(key * 10000000 + i)
    withKeys.sort()
    return [ki % 10000000 for ki in withKeys]
```

Here I combine each number with a left-aligned version of it, for example:

```
     42  =>  42000000000042
   4200  =>  42000000004200
 123456  =>  12345600123456
```

Then just sort these and then extract the lower parts.

Complexity

I think **Time** complexity and **space** complexity are both **O(n)** (at least if sort does what I think it does, I'll check some more), and the space complexity has a low

hidden factor.

The time and memory limits for Python for this problem are pretty low, requiring a fairly efficient solution. On LeetCode, Python ints are 64 bits, so embedding the left-aligned version of numbers in the numbers (solution 2) doesn't cost extra memory. Also, sorting simple ints is fast. Especially since the order from 1 to n is already largely sorted lexicographically, like the streak from 100 to 999 and the streak from 1000 to 9999. And Python's (Tim)sort can take advantage of those streaks and just merge them. If it merges "left to right" like I think it does, then it merges the small streaks first and only integrates the longest streaks last, which leads to overall O(n) time.

Optimizing Solution 2 (accepted in 980, 984, 980 ms)

Instead of assuming that we get numbers up to seven digits long and using constants, this uses the largest power of 10 up to n.

```
def lexicalOrder(self, n):
    highDigit = 1
    while highDigit * 10 <= n:
        highDigit *= 10
    higherDigit = highDigit * 10
    withKeys = []
    for i in xrange(1, n+1):
        key = i
        while key < highDigit:
            key *= 10
        withKeys.append(key * higherDigit + i)
    withKeys.sort()
    return [ki % higherDigit for ki in withKeys]
```

History...

Of course the first thing I had tried was this:

```
def lexicalOrder(self, n):
    return sorted(range(1, n+1), key=str)
```

Outrageously, this wasn't accepted! Got "Memory Limit Exceeded" at input n=49999! So next I tried the `cmp`-version of `sorted` instead of the `key`-version, and building strings only on the fly so it takes less memory:

```
def lexicalOrder(self, n):
    return sorted(range(1, n+1), lambda a, b: cmp(str(a), str(b)))
```

Horrendously, this wasn't accepted! Got "Time Limit Exceeded" at input n=49999! So next I tried converting to strings, sorting those, and converting back to ints. Uses

more memory, but less time:

```
def lexicalOrder(self, n):
    return map(int, sorted(map(str, xrange(1, n+1))))
```

Unfathomably, this wasn't accepted! Got "Memory Limit Exceeded" at input n=49999! The horror! Apparently LeetCode really didn't want me to get away with being lazy. So I tried it without sorting or strings and built the numbers in correct order:

```
def lexicalOrder(self, n):
    def dfs(i):
        if i <= n:
            result.append(i)
            for d in xrange(10):
                dfs(10 * i + d)
    result = []
    for i in range(1, 10):
        dfs(i)
    return result
```

Irritatingly, this wasn't accepted! Got "Time Limit Exceeded"! At input n=**14959**! So it was even **slower** than the above. Geez. And none of this was even remotely close to the "5,000,000" that the problem threatened me with. I gave up. And implemented that last solution in C++. It got accepted.

Later I found out that the "5,000,000" isn't even close to true, the largest actual test case is 49999. But even after lots of different attempts, I still can't get any simple stringify+sort solution accepted. The most efficient I came up with is this:

```
def lexicalOrder(self, n):
    return sorted(xrange(1, n+1), lambda a, b, s=str: 1 if s(b) < s(a) else -1)
```

That uses `xrange`, which is faster than `range`, uses the `cmp`-version of `sorted` because the `key` version gets memory limit exceeded, uses a fast local variable instead of the slower global `str`, and exploits that there are no duplicate numbers so I just have to distinguish two cases which I do with `<` instead of the `cmp` function. Still, after all of that optimization it's not fast enough. But based on comparing it with accepted solutions in custom testing, I think it's close. Maybe 10% too slow.

I did get one stringify+sort solution accepted, but it's less simple. I'll post that one later...

written by StefanPochmann original link here

From Leetcoder.

First Unique Character in a String

Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

Examples:

```
s = "leetcode"
return 0.

s = "loveleetcode",
return 2.
```

Note: You may assume the string contain only lowercase letters.

Solution 1

Hey guys. My solution is pretty straightforward. It takes O(n) and goes through the string twice:

1. Get the frequency of each character.
2. Get the first character that has a frequency of one.

Actually the code below passes all the cases. However, according to@xieta00221, we could change the size of the frequency array to 256 to store other kinds of characters. Thanks for all the other comments and suggestions. Fight on!

```
public class Solution {
    public int firstUniqChar(String s) {
        int freq [] = new int[26];
        for(int i = 0; i < s.length(); i ++)
            freq [s.charAt(i) - 'a'] ++;
        for(int i = 0; i < s.length(); i ++)
            if(freq [s.charAt(i) - 'a'] == 1)
                return i;
        return -1;
    }
}
```

written by ZachC original link here

Solution 2

```
public static int firstUniqChar(String s) {

  char[] a = s.toCharArray();

  for(int i=0; i<a.length;i++){
   if(s.indexOf(a[i])==s.lastIndexOf(a[i])){return i;}
  }
  return -1;
    }
```

written by Lindsayling original link here

Solution 3

```
class Solution {
public:
    int firstUniqChar(string s) {
        int alphabet[26] = {0};
        for (int i = 0; i < s.size(); ++i){++alphabet[s[i] -'a'];}
        int i = 0;
        while (i < s.size() && alphabet[s[i]-'a'] > 1) ++i;
        return i == s.size() ? -1 : i;
    }
};
```

written by AlgoGuruZ original link here

From Leetcoder.

Longest Absolute File Path

Suppose we abstract our file system by a string in the following manner:

The string `"dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext"` represents:

```
dir
    subdir1
    subdir2
        file.ext
```

The directory `dir` contains an empty sub-directory `subdir1` and a sub-directory `subdir2` containing a file `file.ext`.

The string `"dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsubdir2\n\t\t\tfile2.ext"` represents:

```
dir
    subdir1
        file1.ext
        subsubdir1
    subdir2
        subsubdir2
            file2.ext
```

The directory `dir` contains two sub-directories `subdir1` and `subdir2`. `subdir1` contains a file `file1.ext` and an empty second-level sub-directory `subsubdir1`. `subdir2` contains a second-level sub-directory `subsubdir2` containing a file `file2.ext`.

We are interested in finding the longest (number of characters) absolute path to a file within our file system. For example, in the second example above, the longest absolute path is `"dir/subdir2/subsubdir2/file2.ext"`, and its length is `32` (not including the double quotes).

Given a string representing the file system in the above format, return the length of the longest absolute path to file in the abstracted file system. If there is no file in the system, return `0`.

Note:

- The name of a file contains at least a `.` and an extension.
- The name of a directory or sub-directory will not contain a `.`.

Time complexity required: `O(n)` where `n` is the size of the input string.

Notice that `a/aa/aaa/file1.txt` is not the longest file path, if there is another path `aaaaaaaaaaaaaaaaaaaaa/sth.png`.

Solution 1

```
public int lengthLongestPath(String input) {
        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(0); // "dummy" length
        int maxLen = 0;
        for(String s:input.split("\n")){
            int lev = s.lastIndexOf("\t")+1; // number of "\t"
            while(lev+1<stack.size()) stack.pop(); // find parent
            int len = stack.peek()+s.length()-lev+1; // remove "/t", add"/"
            stack.push(len);
            // check if it is file
            if(s.contains(".")) maxLen = Math.max(maxLen, len-1);
        }
        return maxLen;
    }
```

An even shorter and faster solution using array instead of stack:

```
public int lengthLongestPath(String input) {
    String[] paths = input.split("\n");
    int[] stack = new int[paths.length+1];
    int maxLen = 0;
    for(String s:paths){
        int lev = s.lastIndexOf("\t")+1, curLen = stack[lev+1] = stack[lev]+s.len
gth()-lev+1;
        if(s.contains(".")) maxLen = Math.max(maxLen, curLen-1);
    }
    return maxLen;
}
```

written by sky-xu original link here

Solution 2

The number of tabs is my `depth` and for each depth I store the current path length.

```
def lengthLongestPath(self, input):
    maxlen = 0
    pathlen = {0: 0}
    for line in input.splitlines():
        name = line.lstrip('\t')
        depth = len(line) - len(name)
        if '.' in name:
            maxlen = max(maxlen, pathlen[depth] + len(name))
        else:
            pathlen[depth + 1] = pathlen[depth] + len(name) + 1
    return maxlen
```

written by StefanPochmann original link here

Solution 3

```
public:
    int lengthLongestPath(string input) {
        int maxi=0,count=0,ln=1;
        bool isFile=false;
        vector<int> level(200);
        level[0]=0;
        for(int i=0,fin=input.size();i<fin;++i){
            //find which level
            while(input[i]=='\t'){
                ++ln;++i;
            }
            //read file name
            while(input[i]!='\n'&&i<fin){
                if(input[i]=='.')isFile=true;
                ++count;++i;
            }
            //calculate
            if(isFile){
                maxi=max(maxi,level[ln-1]+count);
            }
            else{
                level[ln]=level[ln-1]+count+1;// 1 means '/'
            }
            //reset
            count=0;ln=1;isFile=false;
        }
        return maxi;
    }
};
```

written by XORNOTXOR original link here

From Leetcoder.

Find the Difference

Given two strings ***s*** and ***t*** which consist of only lowercase letters.

String ***t*** is generated by random shuffling string ***s*** and then add one more letter at a random position.

Find the letter that was added in ***t***.

Example:

```
Input:
s = "abcd"
t = "abcde"

Output:
e

Explanation:
'e' is the letter that was added.
```

Solution 1

```
public char findTheDifference(String s, String t) {
 char c = 0;
 for (int i = 0; i < s.length(); ++i) {
  c ^= s.charAt(i);
 }
 for (int i = 0; i < t.length(); ++i) {
  c ^= t.charAt(i);
 }
 return c;
}
```

maybe a more elegant version:

```
public char findTheDifference(String s, String t) {
 int n = t.length();
 char c = t.charAt(n - 1);
 for (int i = 0; i < n - 1; ++i) {
  c ^= s.charAt(i);
  c ^= t.charAt(i);
 }
 return c;
}
```

written by Harvery original link here

Solution 2

It is the same idea with 136. Single Number (https://leetcode.com/problems/single-number/)

```
class Solution {
public:
    char findTheDifference(string s, string t) {
        char r=0;
        for(char c:s) r ^=c;
        for(char c:t) r ^=c;
        return r;
    }
};
```

written by yanchao_hust original link here

Solution 3

Hi. There are several methods you can try to solve this. HashMap, Arrays, Bits, etc. Here, we're going to use a simple array of size 26 for alphabets. Then for each character in **s**, increment the count.

Then for each character in **t**, you should decrement the count. Now if at any point, the count goes below 0, then the character isn't present in **t**

```
        for (int i = 0; i < 26; i++) alpha[i] = 0;
        for (char c : s.toCharArray())
            alpha[ c - 'a' ]++;

        for (char c : t.toCharArray()) {
           //could do decrement first, then check but yeah
            if (--alpha[c - 'a'] < 0)
                return c;
        }

        return 0;
```

written by sharks12 original link here

From Leetcoder.

Perfect Rectangle

Given N axis-aligned rectangles where N > 0, determine if they all together form an exact cover of a rectangular region.

Each rectangle is represented as a bottom-left point and a top-right point. For example, a unit square is represented as [1,1,2,2]. (coordinate of bottom-left point is (1, 1) and top-right point is (2, 2)).

Example 1:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [3,2,4,4],
  [1,3,2,4],
  [2,3,3,4]
]

Return true. All 5 rectangles together form an exac
t cover of a rectangular region.
```

Example 2:

```
rectangles = [
  [1,1,2,3],
  [1,3,2,4],
  [3,1,4,2],
  [3,2,4,4]
]

Return false. Because there is a gap between the tw
o rectangular regions.
```

Example 3:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [3,2,4,4]
]

Return false. Because there is a gap in the top cen
ter.
```

Example 4:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [2,2,4,4]
]
```

Return false. Because two of the rectangles overlap with each other.

Solution 1

This is an expanded version of my earlier post under the contest discussion board. The following code passes through not only the OJ but also various test cases others have pointed out.

Idea

Consider how the corners of all rectangles appear in the large rectangle if there's a perfect rectangular cover.
Rule 1: The local shape of the corner has to follow one of the three following patterns

- Corner of the large rectangle (blue): it occurs only once among all rectangles
- T-junctions (green): it occurs twice among all rectangles
- Cross (red): it occurs four times among all rectangles

Rule 2: A point can only be the top-left corner of at most one sub-rectangle. Similarly it can be the top-right/bottom-left/bottom-right corner of at most one sub-rectangle. Otherwise overlaps occur.

Proof of correctness

Obviously, any perfect cover satisfies the above rules. So the main question is whether there exists an input which satisfy the above rules, yet does not compose a rectangle.

First, ***any overlap is not allowed based on the above rules*** because

- aligned overlap like [[0, 0, 1, 1], [0, 0, 2, 2]] are rejected by Rule 2.
- unaligned overlap will generate a corner in the interior of another sub-rectangle, so it will be rejected by Rule 1.

Second, consider the shape of boundary for the combined shape. The cross pattern does not create boundary. The corner pattern generates a straight angle on the boundary, and the T-junction generates a straight border.
So the shape of the union of rectangles has to be rectangle(s).

Finally, if there are more than two non-overlapping rectangles, at least 8 corners will be found, and cannot be matched to the 4 bounding box corners (be reminded we have shown that there is no chance of overlapping).
So the cover has to be a single rectangle if all above rules are satisfied.

Algorithm

- **Step1:** Based on the above idea we maintain a mapping from (x, y)->mask by scanning the sub-rectangles from beginning to end.
    - (x, y) corresponds to corners of sub-rectangles
    - mask is a 4-bit binary mask. Each bit indicates whether there have been a sub-rectangle with a top-left/top-right/bottom-left/bottom-right corner at (x, y). If we see a conflict while updating mask, it suffice to return a false during the scan.

    - In the meantime we obtain the bounding box of all rectangles (which potentially be the rectangle cover) by getting the upper/lower bound of x/y values.
- **Step 2:** Once the scan is done, we can just browse through the unordered_map to check the whether ***the mask corresponds to a T junction / cross, or corner if it is indeed a bounding box corner***.
  (note: my earlier implementation uses counts of bits in mask to justify corners, and this would not work with certain cases as @StefanPochmann points out).

Complexity

The scan in step 1 is O(n) because it loop through rectangles and inside the loop it updates bounding box and unordered_map in O(1) time.

Step2 visits 1 corner at a time with O(1) computations for at most 4n corners (actually much less because either corner overlap or early stopping occurs). So it's also O(n).

```
// pos encoding: 1 - TL 2- TR 4- BL 8-BR
// return false if a conflict in mask occurs (i.e. there used to be a rectangle wi
th corner (x, y) at pos
inline bool insert_corner(unordered_map<int, unordered_map<int, int>>& corner_cou
nt, int x, int y, int pos) {
    int& m = corner_count[x][y];
    if (m & pos) return false;
    m |= pos;
    return true;
}

bool isRectangleCover(vector<vector<int>>& rectangles) {
    // step 1: counting
    unordered_map<int, unordered_map<int, int>> corner_count;
    int minx = INT_MAX, maxx=INT_MIN, miny=INT_MAX, maxy=INT_MIN;
    for (auto& rect : rectangles) {
        minx = min(minx, rect[0]);
        maxx = max(maxx, rect[2]);
        miny = min(miny, rect[1]);
        maxy = max(maxy, rect[3]);
        if (!insert_corner(corner_count, rect[0], rect[1], 1)) return false;
        if (!insert_corner(corner_count, rect[2], rect[1], 2)) return false;
        if (!insert_corner(corner_count, rect[0], rect[3], 4)) return false;
        if (!insert_corner(corner_count, rect[2], rect[3], 8)) return false;
    }

    //step2: checking
    bool valid_corner[16] = {false};
    bool valid_interior[16] = {false};
    valid_corner[1] = valid_corner[2] = valid_corner[4] = valid_corner[8] = true;
    valid_interior[3] = valid_interior[5] = valid_interior[10] = valid_interior[1
2] = valid_interior[15] = true;

    for (auto itx = corner_count.begin(); itx != corner_count.end(); ++itx) {
        int x = itx->first;
        for (auto ity = itx->second.begin(); ity != itx->second.end(); ++ity) {
            int y = ity->first;
            int mask = ity->second;
            if (((x != minx && x != maxx) || (y != miny && y != maxy)) && !valid_i
nterior[mask])
                return false;
        }
    }
    return true;
}
```

The above code may be refined by changing the 2D unordered_map to 1D. But such improvements has no effect on complexity.

```
struct pairhash {//double hash function for pair key
public:
    template <typename T, typename U>
    size_t operator()(const pair<T, U> &rhs) const {
        size_t l = hash<T>()(rhs.first);
        size_t r = hash<U>()(rhs.second);
        return l + 0x9e3779b9 + (r << 6) + (r >> 2);
    }
};

bool isRectangleCover(vector<vector<int>>& rectangles) {
    // step 1: counting
    unordered_map<pair<int, int>, int, pairhash> corner_count;
    int minx = INT_MAX, maxx=INT_MIN, miny=INT_MAX, maxy=INT_MIN;
    for (auto& rect : rectangles) {
        minx = min(minx, rect[0]);
        maxx = max(maxx, rect[2]);
        miny = min(miny, rect[1]);
        maxy = max(maxy, rect[3]);

        int& m1 = corner_count[make_pair(rect[0], rect[1])];
        if (m1 & 1) return false; else m1 |= 1;
        int& m2 = corner_count[make_pair(rect[2], rect[1])];
        if (m2 & 2) return false; else m2 |= 2;
        int& m3 = corner_count[make_pair(rect[0], rect[3])];
        if (m3 & 4) return false; else m3 |= 4;
        int& m4 = corner_count[make_pair(rect[2], rect[3])];
        if (m4 & 8) return false; else m4 |= 8;
    }

    //step2: checking
    for (const auto& kv: corner_count) {
        pair<int, int> pos; int mask;
        tie(pos, mask) = kv;
        if ((pos.first != minx && pos.first != maxx) || (pos.second != miny && pos
.second != maxy)) {
            if (mask != 3 && mask != 5 && mask != 10 && mask != 12 && mask != 15)
return false;
        }
    }
    return true;
}
```

written by hxtang original link here

Solution 2

The right answer must satisfy two conditions:

1. the large rectangle area should be equal to the sum of small rectangles
2. count of all the points should be even, and that of all the four corner points should be one

'''

```
public boolean isRectangleCover(int[][] rectangles) {
```

```
    if (rectangles.length == 0 || rectangles[0].length == 0) return false;

    int x1 = Integer.MAX_VALUE;
    int x2 = Integer.MIN_VALUE;
    int y1 = Integer.MAX_VALUE;
    int y2 = Integer.MIN_VALUE;

    HashSet<String> set = new HashSet<String>();
    int area = 0;

    for (int[] rect : rectangles) {
        x1 = Math.min(rect[0], x1);
        y1 = Math.min(rect[1], y1);
        x2 = Math.max(rect[2], x2);
        y2 = Math.max(rect[3], y2);

        area += (rect[2] - rect[0]) * (rect[3] - rect[1]);

        String s1 = rect[0] + " " + rect[1];
        String s2 = rect[0] + " " + rect[3];
        String s3 = rect[2] + " " + rect[3];
        String s4 = rect[2] + " " + rect[1];

        if (set.contains(s1)) {
            set.remove(s1);
        } else {
            set.add(s1);
        }
        if (set.contains(s2)) {
            set.remove(s2);
        } else {
            set.add(s2);
        }
        if (set.contains(s3)) {
            set.remove(s3);
        } else {
            set.add(s3);
        }
        if (set.contains(s4)) {
            set.remove(s4);
        } else {
            set.add(s4);
        }
    }

    if (!set.contains(x1 + " " + y1) || !set.contains(x1 + " " + y2) || !set.cont
ains(x2 + " " + y1) || !set.contains(x2 + " " + y2) || set.size() != 4) return fa
lse;

    return area == (x2-x1) * (y2-y1);
}
```

"""

written by hu19 original link here

Solution 3

Standard sweep line solution.
Basic idea:
Sort by x-coordinate.
Insert y-interval into TreeSet, and check if there are intersections.
Delete y-interval.

```
public class Event implements Comparable<Event> {
 int time;
 int[] rect;

 public Event(int time, int[] rect) {
  this.time = time;
  this.rect = rect;
 }

 public int compareTo(Event that) {
  if (this.time != that.time) return this.time - that.time;
  else return this.rect[0] - that.rect[0];
 }
}

public boolean isRectangleCover(int[][] rectangles) {
 PriorityQueue<Event> pq = new PriorityQueue<Event> ();
        // border of y-intervals
 int[] border= {Integer.MAX_VALUE, Integer.MIN_VALUE};
 for (int[] rect : rectangles) {
  Event e1 = new Event(rect[0], rect);
  Event e2 = new Event(rect[2], rect);
  pq.add(e1);
  pq.add(e2);
  if (rect[1] < border[0]) border[0] = rect[1];
  if (rect[3] > border[1]) border[1] = rect[3];
 }
 TreeSet<int[]> set = new TreeSet<int[]> (new Comparator<int[]> () {
  @Override
                // if two y-intervals intersects, return 0
  public int compare (int[] rect1, int[] rect2) {
   if (rect1[3] <= rect2[1]) return -1;
   else if (rect2[3] <= rect1[1]) return 1;
   else return 0;
  }
 });
 int yRange = 0;
 while (!pq.isEmpty()) {
  int time = pq.peek().time;
  while (!pq.isEmpty() && pq.peek().time == time) {
   Event e = pq.poll();
   int[] rect = e.rect;
   if (time == rect[2]) {
    set.remove(rect);
    yRange -= rect[3] - rect[1];
   } else {
    if (!set.add(rect)) return false;
    yRange += rect[3] - rect[1];
```

```
    yRange += rect[3] - rect[1];
    }
   }
                // check intervals' range
   if (!pq.isEmpty() && yRange != border[1] - border[0]) {
                        return false;
    //if (set.isEmpty()) return false;
    //if (yRange != border[1] - border[0]) return false;
   }
  }
  return true;
}
```

written by wddd original link here

From Leetcoder.

Is Subsequence

Given a string **s** and a string **t**, check if **s** is subsequence of **t**.

You may assume that there is only lower case English letters in both **s** and **t**. **t** is potentially a very long (length ~= 500,000) string, and **s** is a short string (

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).

Example 1:
s = `"abc"`, **t** = `"ahbgdc"`

Return `true`.

Example 2:
s = `"axc"`, **t** = `"ahbgdc"`

Return `false`.

Follow up:
If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B, and you want to check one by one to see if T has its subsequence. In this scenario, how would you change your code?

Credits:
Special thanks to @pbrother for adding this problem and creating all test cases.

Subscribe to see which companies asked this question

Solution 1

```
bool isSubsequence(char* s, char* t) {
    while (*t)
        s += *s == *t++;
    return !*s;
}
```

Just go through t and "cross off" the matching characters in s. Then return whether there's nothing left in s.

Sometimes, C makes things easier... here it's helpful that C strings are null terminated and that I can easily pop off a string's first character in constant time just by incrementing the pointer.

written by StefanPochmann original link here

Solution 2

Just use two pointers:

```
public class Solution {
    public boolean isSubsequence(String s, String t) {
        if (s.length() == 0) return true;
        int indexS = 0, indexT = 0;
        while (indexT < t.length()) {
            if (t.charAt(indexT) == s.charAt(indexS)) {
                indexS++;
                if (indexS == s.length()) return true;
            }
            indexT++;
        }
        return false;
    }
}
```

written by sampsonchan original link here

Solution 3

```
def isSubsequence(self, s, t):
    t = iter(t)
    return all(c in t for c in s)
```

Just testing whether all characters in s are also in t (in order).

Based on falsetru's code on StackOverflow which I improved a while ago, see here.

written by StefanPochmann original link here

From Leetcoder.

UTF-8 Validation

A character in UTF8 can be from 1 to 4 bytes long, subjected to the following rules:

1. For 1-byte character, the first bit is a 0, followed by its unicode code.
2. For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followed by n-1 bytes with most significant 2 bits being 10.

This is how the UTF-8 encoding would work:

```
   Char. number range  |        UTF-8 octet sequence
      (hexadecimal)    |              (binary)
   --------------------+---------------------------------------------
   0000 0000-0000 007F | 0xxxxxxx
   0000 0080-0000 07FF | 110xxxxx 10xxxxxx
   0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
   0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

Given an array of integers representing the data, return whether it is a valid utf-8 encoding.

Note:
The input is an array of integers. Only the **least significant 8 bits** of each integer is used to store the data. This means each integer represents only 1 byte of data.

Example 1:

```
data = [197, 130, 1], which represents the octet sequence: 11000101 10000010 00000001.

Return true.
It is a valid utf-8 encoding for a 2-bytes character followed by a 1-byte character.
```

Example 2:

```
data = [235, 140, 4], which represented the octet sequence: 11101011 10001100 00000100.

Return false.
The first 3 bits are all one's and the 4th bit is 0 means it is a 3-bytes character.
The next byte is a continuation byte which starts with 10 and that's correct.
But the second continuation byte does not start with 10, so it is invalid.
```

Subscribe to see which companies asked this question

Solution 1

```
class Solution {
public:
    bool validUtf8(vector<int>& data) {
        int count = 0;
        for (auto c : data) {
            if (count == 0) {
                if ((c >> 5) == 0b110) count = 1;
                else if ((c >> 4) == 0b1110) count = 2;
                else if ((c >> 3) == 0b11110) count = 3;
                else if ((c >> 7)) return false;
            } else {
                if ((c >> 6) != 0b10) return false;
                count--;
            }
        }
        return count == 0;
    }
};
```

written by fight.for.dream original link here

Solution 2

public class Solution {

```
public bool ValidUtf8(int[] data) {
    int bitCount = 0;

    foreach(int n in data){

        if(n >= 192){
            if(bitCount != 0)
                return false;
            else if(n >= 240)
                bitCount = 3;
            else if(n >= 224)
                bitCount = 2;
            else
                bitCount = 1;
        }else if(n >= 128){
            bitCount--;
            if(bitCount < 0)
                return false;
        }else if(bitCount > 0){
            return false;
        }
    }

    return bitCount == 0;
}
```

}

written by lalayangguang original link here

Solution 3

```
class Solution(object):
    def validUtf8(self, data):
        """
        :type data: List[int]
        :rtype: bool
        """
        if len(data) == 0:
            return True
        i = 0
        while i < len(data):
            if data[i] < 128:
                i += 1
            elif data[i] >= 192 and data[i] < 224 and len(data)-i>=2:
                if data[i+1] >= 128 and data[i+1] < 192:
                    i += 2
                else:
                    return False
            elif data[i] >= 224 and data[i] < 240 and len(data)-i>=3:
                if data[i+1] >= 128 and data[i+1] < 192 and data[i+2] >= 128 and
data[i+2] < 192:
                    i += 3
                else:
                    return False
            elif data[i] >= 240 and data[i] < 248 and len(data)-i>=4:
                if data[i+1] >= 128 and data[i+1] < 192 and data[i+2] >= 128 and
data[i+2] < 192 and data[i+3] >= 128 and data[i+3] < 192:
                    i += 4
                else:
                    return False
            else:
                return False
        return True
```

written by AlgoGuruZ original link here

From Leetcoder.

Decode String

Given an encoded string, return it's decoded string.

The encoding rule is: `k[encoded_string]`, where the *encoded_string* inside the square brackets is being repeated exactly *k* times. Note that *k* is guaranteed to be a positive integer.

You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, *k*. For example, there won't be input like `3a` or `2[4]`.

Examples:

```
s = "3[a]2[bc]", return "aaabcbc".
s = "3[a2[c]]", return "accaccacc".
s = "2[abc]3[cd]ef", return "abcabccdcdcdef".
```

Subscribe to see which companies asked this question

Solution 1

```
class Solution(object):
    def decodeString(self, s):
        stack = []
        stack.append(["", 1])
        num = ""
        for ch in s:
            if ch.isdigit():
              num += ch
            elif ch == '[':
                stack.append(["", int(num)])
                num = ""
            elif ch == ']':
                st, k = stack.pop()
                stack[-1][0] += st*k
            else:
                stack[-1][0] += ch
        return stack[0][0]
```

written by ayushpatwari original link here

Solution 2

```
class Solution {
public:
    string decodeString(string s, int& i) {
        string res;

        while (i < s.length() && s[i] != ']') {
            if (!isdigit(s[i]))
                res += s[i++];
            else {
                int n = 0;
                while (i < s.length() && isdigit(s[i]))
                    n = n * 10 + s[i++] - '0';

                i++; // '['
                string t = decodeString(s, i);
                i++; // ']'

                while (n-- > 0)
                    res += t;
            }
        }

        return res;
    }

    string decodeString(string s) {
        int i = 0;
        return decodeString(s, i);
    }
};
```

written by bluedawnstar original link here

Solution 3

```java
public class Solution {
    public String decodeString(String s) {
        String res = "";
        Stack<Integer> countStack = new Stack<>();
        Stack<String> resStack = new Stack<>();
        int idx = 0;
        while (idx < s.length()) {
            if (Character.isDigit(s.charAt(idx))) {
                int count = 0;
                while (Character.isDigit(s.charAt(idx))) {
                    count = 10 * count + (s.charAt(idx) - '0');
                    idx++;
                }
                countStack.push(count);
            }
            else if (s.charAt(idx) == '[') {
                resStack.push(res);
                res = "";
                idx++;
            }
            else if (s.charAt(idx) == ']') {
                StringBuilder temp = new StringBuilder (resStack.pop());
                int repeatTimes = countStack.pop();
                for (int i = 0; i < repeatTimes; i++) {
                    temp.append(res);
                }
                res = temp.toString();
                idx++;
            }
            else {
                res += s.charAt(idx++);
            }
        }
        return res;
    }
}
```

written by sampsonchan original link here

From Leetcoder.

Longest Substring with At Least K Repeating Characters

Find the length of the longest substring T of a given string (consists of lowercase letters only) such that every character in T appears no less than k times.

Example 1:

```
Input:
s = "aaabb", k = 3

Output:
3

The longest substring is "aaa", as 'a' is repeated 3 times.
```

Example 2:

```
Input:
s = "ababbc", k = 2

Output:
5

The longest substring is "ababb", as 'a' is repeated 2 times and 'b' is repeated 3
times.
```

Subscribe to see which companies asked this question

Solution 1

Update:

As pointed out by @hayleyhu, I can just take the first too rare character instead of a rarest. Submitted once, accepted in 48 ms.

```
def longestSubstring(self, s, k):
    for c in set(s):
        if s.count(c) < k:
            return max(self.longestSubstring(t, k) for t in s.split(c))
    return len(s)
```

Original:

```
def longestSubstring(self, s, k):
    if len(s) < k:
        return 0
    c = min(set(s), key=s.count)
    if s.count(c) >= k:
        return len(s)
    return max(self.longestSubstring(t, k) for t in s.split(c))
```

If every character appears at least k times, the whole string is ok. Otherwise split by a least frequent character (because it will always be too infrequent and thus can't be part of any ok substring) and make the most out of the splits.

As usual for Python here, the runtime varies a lot, this got accepted in times from 32 ms to 74 ms.

written by StefanPochmann original link here

Solution 2

```
public int longestSubstring(String s, int k) {
    char[] str = s.toCharArray();
    return helper(str,0,s.length(),k);
}
private int helper(char[] str, int start, int end,  int k){
    if(end<start) return 0;
    if(end-start<k) return 0;//substring length shorter than k.
    int[] count = new int[26];
    for(int i = start;i<end;i++){
        int idx = str[i]-'a';
        count[idx]++;
    }
    for(int i = 0;i<26;i++){
        if(count[i]==0)continue;//i+'a' does not exist in the string, skip it.
        if(count[i]<k){
            for(int j = start;j<end;j++){
                if(str[j]==i+'a'){
                    int left = helper(str,start,j,k);
                    int right = helper(str,j+1,end,k);
                    return Math.max(left,right);
                }
            }
        }
    }
    return end-start;
}
```

written by Nakanu original link here

Solution 3

Sol1: a simple improvement on the naive quaratic solution. The idea is that if a locally longest substr is found, there's no need to check substrs overlapping it.
Sol1 can run O(n) times in some cases, but worst case is O(n2). Anyway the C++ run time is 3ms.

```
int longestSubstring(string s, int k) {
   int max_len = 0;
   for (int first = 0; first+k <= s.size();) {
       int count[26] = {0};
       int mask = 0;
       int max_last = first;
       for (int last = first; last < s.size(); ++last) {
           int i = s[last] - 'a';
           count[i]++;
           if (count[i]<k) mask |= (1 << i);
           else   mask &= (~(1 << i));

           if (mask == 0) {
               max_len = max(max_len, last-first+1);
               max_last = last;
           }
       }
       first = max_last + 1;
   }
   return max_len;
}
```

Sol2: recursive: split the string into substrs by characters of occurrence less than k. Then recursively apply the problem to each substr.
Worst case of Sol2 is O(n), because there are at most 26 levels of recursions. The C++ impl. runs 6ms. I suspect this is because the current test cases does not cover enough cases in favor of this solution in run time.

```
int longestSubstring(string s, int k) {
    return longestSubstring_recur(s, k, 0, s.size());
}

int longestSubstring_recur(const string& s, int k, int first, int last) {
    int count[26] = {0};
    for (int j = first; j < last; ++j) ++count[s[j] - 'a'];

    int max_len = 0;
    for (int j = first; j < last;) {
        while (j < last && count[s[j]-'a']<k) ++j;
        if (j == last) break;
        int l = j;
        while (l < last && count[s[l]-'a']>=k) ++l;
        //all chars appear more than k times
        if (j == first && l == last) return last-first;
        max_len = max(max_len, longestSubstring_recur(s, k, j, l));
        j = l;
    }
    return max_len;
}
```

written by hxtang original link here

From Leetcoder.

Rotate Function

Given an array of integers `A` and let n to be its length.

Assume B_k to be an array obtained by rotating the array `A` k positions clock-wise, we define a "rotation function" `F` on `A` as follow:

$F(k) = 0 * B_k[0] + 1 * B_k[1] + ... + (n-1) * B_k[n-1]$.

Calculate the maximum value of `F(0), F(1), ..., F(n-1)`.

Note:
n is guaranteed to be less than 10^5.

Example:

```
A = [4, 3, 2, 6]

F(0) = (0 * 4) + (1 * 3) + (2 * 2) + (3 * 6) = 0 + 3 + 4 + 18 = 25
F(1) = (0 * 6) + (1 * 4) + (2 * 3) + (3 * 2) = 0 + 4 + 6 + 6 = 16
F(2) = (0 * 2) + (1 * 6) + (2 * 4) + (3 * 3) = 0 + 6 + 8 + 9 = 23
F(3) = (0 * 3) + (1 * 2) + (2 * 6) + (3 * 4) = 0 + 2 + 12 + 12 = 26

So the maximum value of F(0), F(1), F(2), F(3) is F(3) = 26.
```

Subscribe to see which companies asked this question

Solution 1

```
F(k) = 0 * Bk[0] + 1 * Bk[1] + ... + (n-1) * Bk[n-1]
F(k-1) = 0 * Bk-1[0] + 1 * Bk-1[1] + ... + (n-1) * Bk-1[n-1]
       = 0 * Bk[1] + 1 * Bk[2] + ... + (n-2) * Bk[n-1] + (n-1) * Bk[0]
```

Then,

```
F(k) - F(k-1) = Bk[1] + Bk[2] + ... + Bk[n-1] + (1-n)Bk[0]
              = (Bk[0] + ... + Bk[n-1]) - nBk[0]
              = sum - nBk[0]
```

Thus,

```
F(k) = F(k-1) + sum - nBk[0]
```

What is Bk[0]?

```
k = 0; B[0] = A[0];
k = 1; B[0] = A[len-1];
k = 2; B[0] = A[len-2];
...
```

```
int allSum = 0;
int len = A.length;
int F = 0;
for (int i = 0; i < len; i++) {
    F += i * A[i];
    allSum += A[i];
}
int max = F;
for (int i = len - 1; i >= 1; i--) {
    F = F + allSum - len * A[i];
    max = Math.max(F, max);
}
return max;
```

written by oreomilkshake original link here

Solution 2

Consider we have 5 coins A,B,C,D,E

According to the problem statement
F(0) = (0*A*) + (*1*B) + (2*C*) + (*3*D) + (4*E*)
F(1) = (*4*A) + (0*B*) + (*1*C) + (2*D*) + (*3*E)
F(2) = (3*A*) + (*4*B) + (0*C*) + (*1*D) + (2*E)

This problem at a glance seem like a difficult problem. I am not very strong in mathematics, so this is how I visualize this problem

We can construct F(1) from F(0) by two step:
Step 1. taking away one count of each coin from F(0), this is done by subtracting "sum" from "iteration" in the code below
after step 1 F(0) = (-1*A*) + (*0*B) + (1*C*) + (*2*D) + (3*E)

Step 2. Add n times the element which didn't contributed in F(0), which is A. This is done by adding "A[j-1]*len" in the code below.*
after step 2 F(0) = (*4*A) + (0*B*) + (*1*C) + (2*D*) + (*3*E)

At this point F(0) can be considered as F(1) and F(2) to F(4) can be constructed by repeating the above steps.

Hope this explanation helps, cheers!

```
    public int maxRotateFunction(int[] A) {
        if(A.length == 0){
            return 0;
        }

        int sum =0, iteration = 0, len = A.length;

        for(int i=0; i<len; i++){
            sum += A[i];
            iteration += (A[i] * i);
        }

        int max = iteration;
        for(int j=1; j<len; j++){
            // for next iteration lets remove one entry value of each entry and t
he prev 0 * k
            iteration = iteration - sum + A[j-1]*len;
            max = Math.max(max, iteration);
        }

        return max;
    }
```

written by chiranjeeb2 original link here

Solution 3

```
public class Solution {

 public int maxRotateFunction(int[] A) {
  int n = A.length;
  int sum = 0;
  int candidate = 0;

  for (int i = 0; i < n; i++) {
   sum += A[i];
   candidate += A[i] * i;
  }
  int best = candidate;

  for (int i = n - 1; i > 0; i--) {
   candidate = candidate + sum - A[i] * n;
   best = Math.max(best, candidate);
  }
  return best;
 }
}
```

written by lzb700m original link here

From Leetcoder.

Integer Replacement

Given a positive integer n and you can do operations as follow:

1. If n is even, replace n with `n/2`.
2. If n is odd, you can replace n with either `n + 1` or `n - 1`.

What is the minimum number of replacements needed for n to become 1?

Example 1:

```
Input:
8

Output:
3

Explanation:
8 -> 4 -> 2 -> 1
```

Example 2:

```
Input:
7

Output:
4

Explanation:
7 -> 8 -> 4 -> 2 -> 1
or
7 -> 6 -> 3 -> 2 -> 1
```

Subscribe to see which companies asked this question

Solution 1

I really think it should be tagged medium because there are many subtleties and good understanding of binary arithmetic is required.

The first step towards solution is to realize that you're allowed to remove the LSB only if it's zero. And to reach the target as fast as possible, removing digits is the best way to go. Hence, even numbers are better than odd. This is quite obvious.

What is not so obvious is what to do with odd numbers. One may think that you just need to remove as many 1's as possible to increase the evenness of the number. Wrong! Look at this example:

```
111011 -> 111010 -> 11101 -> 11100 -> 1110 -> 111 -> 1000 -> 100 -> 10 -> 1
```

And yet, this is not the best way because

```
111011 -> 111100 -> 11110 -> 1111 -> 10000 -> 1000 -> 100 -> 10 -> 1
```

See? Both `111011 -> 111010` and `111011 -> 111100` remove the same number of 1's, but the second way is better.

So, we just need to remove as many 1's as possible, doing +1 in case of a tie? Not quite. The infamous test with n=3 fails for that strategy because `11 -> 10 -> 1` is better than `11 -> 100 -> 10 -> 1`. Fortunately, that's the only exception (or at least I can't think of any other, and there are none in the tests).

So the logic is:

1. If `n` is even, halve it.
2. If `n=3` or `n-1` has less 1's than `n+1`, decrement `n`.
3. Otherwise, increment `n`.

Here is an example of such a solution in Java:

```java
public int integerReplacement(int n) {
    int c = 0;
    while (n != 1) {
        if ((n & 1) == 0) {
            n >>>= 1;
        } else if (n == 3 || Integer.bitCount(n + 1) > Integer.bitCount(n - 1)) {
            --n;
        } else {
            ++n;
        }
        ++c;
    }
    return c;
}
```

Of course, doing `bitCount` on every iteration is not the best way. It is enough to examine the last two digits to figure out whether incrementing or decrementing will

give more 1's. Indeed, if a number ends with 01, then certainly decrementing is the way to go. Otherwise, if it ends with 11, then certainly incrementing is at least as good as incrementing (`*011 -> *010 / *100`) or even better (if there are three or more 1's). This leads to the following solution:

```
public int integerReplacement(int n) {
    int c = 0;
    while (n != 1) {
        if ((n & 1) == 0) {
            n >>>= 1;
        } else if (n == 3 || ((n >>> 1) & 1) == 0) {
            --n;
        } else {
            ++n;
        }
        ++c;
    }
    return c;
}
```

An alternative approach to intuitive algorithm was very well put by@dettier in a discussion: you should create as many trailing zeroes as you can. This way you can avoid the tie-breaking trap (there can be no ties), but you'll still have to handle the n=3 exception separately.

written by SergeyTachenov original link here

Solution 2

When n is even, the operation is fixed. The only unknown procedure is when it is odd. When n is odd it can be written into the form n = 2k+1 (k is a non-negative integer.). That is, n+1 = 2k+2 and n-1 = 2k. Then, (n+1)/2 = k+1 and (n-1)/2 = k. So the one of (n+1)/2 and (n-1)/2 is even, another is odd. And the "best" case of this problem is to divide as much as possible. Because of that, always pick n+1 or n-1 based on if it can be divided by 4. The only special case of that is when n=3 you would like to pick n-1 rather than n+1.

```
public int integerReplacement(int n) {
    if(n==Integer.MAX_VALUE) return 32; //n = 2^31-1;
    int count = 0;
    while(n>1){
        if(n%2==0) n/=2;
        else{
            if((n+1)%4==0&&(n-1!=2)) n+=1;
            else n-=1;
        }
        count++;
    }
    return count;
}
```

written by Nakanu original link here

Solution 3

For this problem, if we look at the binary form of each number, we can get the idea that for each '1' (except for the first '1') it counts to two steps, for each '0', it counts to one step.
So our goal is to use +1 or -1 to reduce steps.

For example,
13 = 1101
If we plus one, we can get 1110; if we reduce one, we can get 1100;
1110 needs 2+2+1 = 5 steps, while 1100 only needs 2+1+1 = 4 steps, so we choose n-1 in this step.

Use long to avoid overflow (if n is Integer.MAX_VALUE).

```
public class Solution {
    public int integerReplacement(int n) {
        long N = n;
        long small,big;
        int cnt = 0;
        while( N != 1){
         small = (N  & ( N -1));
         big = ( N & (N + 1));
         if( (N & 1) == 0){
          N >>= 1;
         }
         else if ( (small & (small-1)) <= (big & (big-1))){
          N = N - 1;
         }
         else{
          N = N +1;
         }
         cnt++;
        }
        return cnt;
    }
}
```

written by ZebraRabbit original link here

Random Pick Index

Given an array of integers with possible duplicates, randomly output the index of a given target number. You can assume that the given target number must exist in the array.

Note:
The array size can be very large. Solution that uses too much extra space will not pass the judge.

Example:

```
int[] nums = new int[] {1,2,3,3,3};
Solution solution = new Solution(nums);

// pick(3) should return either index 2, 3, or 4 randomly. Each index should have e
qual probability of returning.
solution.pick(3);

// pick(1) should return 0. Since in the array only nums[0] is equal to 1.
solution.pick(1);
```

Subscribe to see which companies asked this question

Solution 1

```
public class Solution {

    int[] nums;
    Random rnd;

    public Solution(int[] nums) {
        this.nums = nums;
        this.rnd = new Random();
    }

    public int pick(int target) {
        int result = -1;
        int count = 0;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] != target)
                continue;
            if (rnd.nextInt(++count) == 0)
                result = i;
        }

        return result;
    }
}```
```

written by dettier original link here

Solution 2

Because I've made a rather naive map-of-index-lists Java solution and it was happily accepted by the OJ. So far I see three types of solutions:

1. Like mine, O(N) memory, O(N) init, O(1) pick.
2. Like @dettier's Reservoir Sampling. O(1) init, O(1) memory, but O(N) to pick.
3. Like @chin-heng's binary search: O(N) memory, O(N lg N) init, O(lg N) pick.

Are all three kinds acceptable?

written by SergeyTachenov original link here

Solution 3

```
class Solution {
public:
    vector<int> n;
    Solution(vector<int> nums)
    {
        n = nums;
    }

    int pick(int target)
    {
        int count = 0, res = -1;
        for (int i = 0; i < n.size(); ++i)
        {
            if(n[i] != target) continue;
            if(++count == 1) res = i;
            else
                if(!(rand()%count)) res = i;
        }
        return res;
    }
};
```

written by ycf303 original link here

From Leetcoder.

Evaluate Division

Equations are given in the format `A / B = k`, where `A` and `B` are variables represented as strings, and `k` is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return `-1.0`.

Example:
Given `a / b = 2.0, b / c = 3.0.`
queries are: `a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? .`
return `[6.0, 0.5, -1.0, 1.0, -1.0 ].`

The input is: `vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries`, where `equations.size() == values.size()`, and the values are positive. This represents the equations. Return `vector<double>`.

According to the example above:

```
equations = [ ["a", "b"], ["b", "c"] ],
values = [2.0, 3.0],
queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].
```

The input is always valid. You may assume that evaluating the queries will result in no division by zero and there is no contradiction.

Subscribe to see which companies asked this question

Solution 1

Image a/b = k as a link between node a and c, the weight from a to c is k, the reverse link is 1/k. Query is to find a path between two node.

```java
    public double[] calcEquation(String[][] equations, double[] values, String[][
] queries) {
        HashMap<String, ArrayList<String>> pairs = new HashMap<String, ArrayList<
String>>();
        HashMap<String, ArrayList<Double>> valuesPair = new HashMap<String, Array
List<Double>>();
        for (int i = 0; i < equations.length; i++) {
            String[] equation = equations[i];
            if (!pairs.containsKey(equation[0])) {
                pairs.put(equation[0], new ArrayList<String>());
                valuesPair.put(equation[0], new ArrayList<Double>());
            }
            if (!pairs.containsKey(equation[1])) {
                pairs.put(equation[1], new ArrayList<String>());
                valuesPair.put(equation[1], new ArrayList<Double>());
            }
            pairs.get(equation[0]).add(equation[1]);
            pairs.get(equation[1]).add(equation[0]);
            valuesPair.get(equation[0]).add(values[i]);
            valuesPair.get(equation[1]).add(1/values[i]);
        }

        double[] result = new double[queries.length];
        for (int i = 0; i < queries.length; i++) {
            String[] query = queries[i];
            result[i] = dfs(query[0], query[1], pairs, valuesPair, new HashSet<St
ring>(), 1.0);
            if (result[i] == 0.0) result[i] = -1.0;
        }
        return result;
    }

    private double dfs(String start, String end, HashMap<String, ArrayList<String
>> pairs, HashMap<String, ArrayList<Double>> values, HashSet<String> set, double
value) {
        if (set.contains(start)) return 0.0;
        if (!pairs.containsKey(start)) return 0.0;
        if (start.equals(end)) return value;
        set.add(start);

        ArrayList<String> strList = pairs.get(start);
        ArrayList<Double> valueList = values.get(start);
        double tmp = 0.0;
        for (int i = 0; i < strList.size(); i++) {
            tmp = dfs(strList.get(i), end, pairs, values, set, value*valueList.ge
t(i));
            if (tmp != 0.0) {
                break;
            }
        }
        set.remove(start);
        return tmp;
    }
```

written by helloc93 original link here

Solution 2

A variation of **Floyd–Warshall**, computing quotients instead of shortest paths. An equation `A/B=k` is like a graph edge `A->B`, and `(A/B)*(B/C)*(C/D)` is like the path `A->B->C->D`. Submitted once, accepted in 35 ms.

```
def calcEquation(self, equations, values, queries):
    quot = collections.defaultdict(dict)
    for (num, den), val in zip(equations, values):
        quot[num][num] = quot[den][den] = 1.0
        quot[num][den] = val
        quot[den][num] = 1 / val
    for k, i, j in itertools.permutations(quot, 3):
        if k in quot[i] and j in quot[k]:
            quot[i][j] = quot[i][k] * quot[k][j]
    return [quot[num].get(den, -1.0) for num, den in queries]
```

Variation without the `if` (submitted twice, accepted in 68 and 39 ms):

```
def calcEquation(self, equations, values, queries):
    quot = collections.defaultdict(dict)
    for (num, den), val in zip(equations, values):
        quot[num][num] = quot[den][den] = 1.0
        quot[num][den] = val
        quot[den][num] = 1 / val
    for k in quot:
        for i in quot[k]:
            for j in quot[k]:
                quot[i][j] = quot[i][k] * quot[k][j]
    return [quot[num].get(den, -1.0) for num, den in queries]
```

Could save a line with `for i, j in itertools.permutations(quot[k], 2)` but it's longer and I don't like it as much here.

written by StefanPochmann original link here

Solution 3

```
class Solution
{
    // date: 2016-09-12     location: Santa Clara City Library
public:
    vector<double> calcEquation(vector<pair<string, string>> equations, vector<do
uble>& values, vector<pair<string, string>> queries)
    {
        unordered_map<string, Node*> map;
        vector<double> res;
        for (int i = 0; i < equations.size(); i ++)
        {
            string s1 = equations[i].first, s2 = equations[i].second;
            if (map.count(s1) == 0 && map.count(s2) == 0)
            {
                map[s1] = new Node();
                map[s2] = new Node();
                map[s1] -> value = values[i];
                map[s2] -> value = 1;
                map[s1] -> parent = map[s2];
            }
            else if (map.count(s1) == 0)
            {
                map[s1] = new Node();
                map[s1] -> value = map[s2] -> value * values[i];
                map[s1] -> parent = map[s2];
            }
            else if (map.count(s2) == 0)
            {
                map[s2] = new Node();
                map[s2] -> value = map[s1] -> value / values[i];
                map[s2] -> parent = map[s1];
            }
            else
                unionNodes(map[s1], map[s2], values[i], map);
        }

        for (auto query : queries)
        {
            if (map.count(query.first) == 0 || map.count(query.second) == 0 || fi
ndParent(map[query.first]) != findParent(map[query.second]))
                res.push_back(-1);
            else
                res.push_back(map[query.first] -> value / map[query.second] -> va
lue);
        }
        return res;
    }

private:
    struct Node
    {
        Node* parent;
        double value = 0.0;
        Node()  {parent = this;}
```

```
    };

    void unionNodes(Node* node1, Node* node2, double num, unordered_map<string, N
ode*>& map)
    {
        Node* parent1 = findParent(node1), *parent2 = findParent(node2);
        double ratio = node2 -> value * num / node1 -> value;
        for (auto it = map.begin(); it != map.end(); it ++)
            if (findParent(it -> second) == parent1)
                it -> second -> value *= ratio;
        parent1 -> parent = parent2;
    }

    Node* findParent(Node* node)
    {
        if (node -> parent == node)
            return node;
        node -> parent = findParent(node -> parent);
        return node -> parent;
    }
};
```

written by Mad_air original link here

From Leetcoder.

Nth Digit

Find the nth digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

Note:
n is positive and will fit within the range of a 32-bit signed integer (n 31).

Example 1:

```
Input:
3

Output:
3
```

Example 2:

```
Input:
11

Output:
0

Explanation:
The 11th digit of the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... is a 0, which
 is part of the number 10.
```

Subscribe to see which companies asked this question

Solution 1

Straight forward way to solve the problem in 3 steps:

1. find the length of the number where the nth digit is from
2. find the actual number where the nth digit is from
3. find the nth digit and return

```
public int findNthDigit(int n) {
 int len = 1;
 long count = 9;
 int start = 1;

 while (n > len * count) {
  n -= len * count;
  len += 1;
  count *= 10;
  start *= 10;
 }

 start += (n - 1) / len;
 String s = Integer.toString(start);
 return Character.getNumericValue(s.charAt((n - 1) % len));
}
```

written by lzb700m original link here

Solution 2

Check the same-length ranges 1-9, 10-99, 100-999, 1000-9999, etc.

Python:

```python
def findNthDigit(self, n):
    n -= 1
    for digits in range(1, 11):
        first = 10**(digits - 1)
        if n < 9 * first * digits:
            return int(str(first + n/digits)[n%digits])
        n -= 9 * first * digits
```

Java:

```java
public int findNthDigit(int n) {
    n -= 1;
    int digits = 1, first = 1;
    while (n / 9 / first / digits >= 1) {
        n -= 9 * first * digits;
        digits++;
        first *= 10;
    }
    return (first + n/digits + "").charAt(n%digits) - '0';
}
```

Using divisions instead of multiplications to prevent overflow.

written by StefanPochmann original link here

Solution 3

```
We are tightening the range of our target gradually.
1. Is this number that has nth digit between 100~999, 1000~9999 or some other rang
e?
2. What is this number?
3. Within thin number, which digit is the nth digit?

number of the digits at each "level"
1-9: 9 digits
10-99 : 90 * 2 = 180 digits
100-999 : 900 * 3 = 2700 digits
1000-9999 : 9000 * 4 = 36000 digits
....
*/
public class Solution {
    public int findNthDigit(int n) {
        if(n < 1) return 0;
        if(n < 10) return n;
        int counter = 1;  //stores the level number
        int base = 0;      //stores the biggest number from previous level
        while(n > (9 * Math.pow(10,counter -1) * (counter))){
            base += 9 * Math.pow(10,counter -1);
            n -= (9 * Math.pow(10,counter -1) * (counter));
            counter++;
        }
        //target is the actual number that has nth digit
        int target = base + ((n + counter - 1) / counter);  //to get the ceiling
of n / counter
        int offset = n % counter;
        offset = (offset == 0) ? 0:counter - offset;
        for(int i = 0; i < offset; i++){
            target = target / 10;
        }
        return target % 10;
    }
}
```

written by DonaldTrump original link here

From Leetcoder.

Binary Watch

A binary watch has 4 LEDs on the top which represent the hours (0-11), and the 6 LEDs on the bottom represent the minutes (0-59).

Each LED represents a zero or one, with the least significant bit on the right.

For example, the above binary watch reads "3:25".

Given a non-negative integer *n* which represents the number of LEDs that are currently on, return all possible times the watch could represent.

Example:

```
Input: n = 1
Return: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04", "0:08", "0:16", "0
:32"]
```

Note:

- The order of output does not matter.
- The hour must not contain a leading zero, for example "01:00" is not valid, it should be "1:00".
- The minute must be consist of two digits and may contain a leading zero, for example "10:2" is not valid, it should be "10:02".

Subscribe to see which companies asked this question

Solution 1

Just go through the possible times and collect those with the correct number of one-bits.

Python:

```
def readBinaryWatch(self, num):
    return ['%d:%02d' % (h, m)
            for h in range(12) for m in range(60)
            if (bin(h) + bin(m)).count('1') == num]
```

Java:

```
public List<String> readBinaryWatch(int num) {
    List<String> times = new ArrayList<>();
    for (int h=0; h<12; h++)
        for (int m=0; m<60; m++)
            if (Integer.bitCount(h * 64 + m) == num)
                times.add(String.format("%d:%02d", h, m));
    return times;
}
```

written by StefanPochmann original link here

Solution 2

```
vector<string> readBinaryWatch(int num) {
    vector<string> rs;
    for (int h = 0; h < 12; h++)
    for (int m = 0; m < 60; m++)
        if (bitset<10>(h << 6 | m).count() == num)
            rs.emplace_back(to_string(h) + (m < 10 ? ":0" : ":") + to_string(m));
    return rs;
}
```

written by mzchen original link here

Solution 3

```
public class Solution {
    public List<String> readBinaryWatch(int num) {
        List<String> res = new ArrayList<>();
        int[] nums1 = new int[]{8, 4, 2, 1}, nums2 = new int[]{32, 16, 8, 4, 2,
1};
        for(int i = 0; i <= num; i++) {
            List<Integer> list1 = generateDigit(nums1, i);
            List<Integer> list2 = generateDigit(nums2, num - i);
            for(int num1: list1) {
                if(num1 >= 12) continue;
                for(int num2: list2) {
                    if(num2 >= 60) continue;
                    res.add(num1 + ":" + (num2 < 10 ? "0" + num2 : num2));
                }
            }
        }
        return res;
    }

    private List<Integer> generateDigit(int[] nums, int count) {
        List<Integer> res = new ArrayList<>();
        generateDigitHelper(nums, count, 0, 0, res);
        return res;
    }

    private void generateDigitHelper(int[] nums, int count, int pos, int sum, Lis
t<Integer> res) {
        if(count == 0) {
            res.add(sum);
            return;
        }

        for(int i = pos; i < nums.length; i++) {
            generateDigitHelper(nums, count - 1, i + 1, sum + nums[i], res);
        }
    }
}
```

written by xietao0221 original link here

From Leetcoder.

Remove K Digits

Given a non-negative integer *num* represented as a string, remove *k* digits from the number so that the new number is the smallest possible.

Note:

- The length of *num* is less than 10^5 and will be $\geq k$.
- The given *num* does not contain any leading zero.

Example 1:

```
Input: num = "1432219", k = 3
Output: "1219"
Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which
is the smallest.
```

Example 2:

```
Input: num = "10200", k = 1
Output: "200"
Explanation: Remove the leading 1 and the number is 200. Note that the output must
not contain leading zeroes.
```

Example 3:

```
Input: num = "10", k = 2
Output: "0"
Explanation: Remove all the digits from the number and it is left with nothing whic
h is 0.
```

Subscribe to see which companies asked this question

Solution 1

O(n) Solution

Go through the digits from left to right, remove previous digits if that makes the number smaller (and if we still have to remove digits). Almost the same as the `prep` function from my solution to an earlier problem, which did basically the same except it **maximized** the number.

```
def removeKdigits(self, num, k):
    out = []
    for d in num:
        while k and out and out[-1] > d:
            out.pop()
            k -= 1
        out.append(d)
    return ''.join(out[:-k or None]).lstrip('0') or '0'
```

Regex Solution

k times remove the leftmost digit followed by a smaller digit (or remove the last digit). Didn't think this would be fast enough, but it is :-)

```
def removeKdigits(self, num, k):
    sub = re.compile('1[0]|2[01]|3[0-2]|4[0-3]|5[0-4]|6[0-5]|7[0-6]|8[0-7]|9[0-8]
|.$').sub
    for _ in range(k):
        num = sub(lambda m: m.group()[1:], num, 1)
    return num.lstrip('0') or '0'
```

written by StefanPochmann original link here

Solution 2

```java
public class Solution {
    public String removeKdigits(String num, int k) {
        int digits = num.length() - k;
        char[] stk = new char[num.length()];
        int top = 0;
        // k keeps track of how many characters we can remove
        // if the previous character in stk is larger than the current one
        // then removing it will get a smaller number
        // but we can only do so when k is larger than 0
        for (int i = 0; i < num.length(); ++i) {
            char c = num.charAt(i);
            while (top > 0 && stk[top-1] > c && k > 0) {
                top -= 1;
                k -= 1;
            }
            stk[top++] = c;
        }
        // find the index of first non-zero digit
        int idx = 0;
        while (idx < digits && stk[idx] == '0') idx++;
        return idx == digits? "0": new String(stk, idx, digits - idx);
    }
}
```

written by keyulai92gmail.com original link here

Solution 3

k is the number of char needs dropping, 'keep' is the number of char that should keep.
We need to remove the beginning zeroes, and finally check if the result is empty of not.

```
class Solution {
public:
    string removeKdigits(string num, int k) {
        string res = "";
        int n = num.size(), keep = n - k;
        for (char c : num) {
            while (k && res.size() && res.back() > c) {
                res.pop_back();
                --k;
            }
            res.push_back(c);
        }
        res.resize(keep);
        while (!res.empty() && res[0] == '0') res.erase(res.begin());
        return res.empty() ? "0" : res;
    }
};
```

written by grandyang original link here

From Leetcoder.

Frog Jump

A frog is crossing a river. The river is divided into x units and at each unit there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of stones' positions (in units) in sorted ascending order, determine if the frog is able to cross the river by landing on the last stone. Initially, the frog is on the first stone and assume the first jump must be 1 unit.

If the frog has just jumped k units, then its next jump must be either k - 1, k, or k + 1 units. Note that the frog can only jump in the forward direction.

Note:

- The number of stones is ≥ 2 and is
- Each stone's position will be a non-negative integer 31.
- The first stone's position is always 0.

Example 1:

```
[0,1,3,5,6,8,12,17]

There are a total of 8 stones.
The first stone at the 0th unit, second stone at the 1st unit,
third stone at the 3rd unit, and so on...
The last stone at the 17th unit.

Return true. The frog can jump to the last stone by jumping
1 unit to the 2nd stone, then 2 units to the 3rd stone, then
2 units to the 4th stone, then 3 units to the 6th stone,
4 units to the 7th stone, and 5 units to the 8th stone.
```

Example 2:

```
[0,1,2,3,4,8,9,11]

Return false. There is no way to jump to the last stone as
the gap between the 5th and 6th stone is too large.
```

Subscribe to see which companies asked this question

Solution 1

Search for the last stone in a depth-first way, prune those exceeding the [k-1,k+1] range. Well, I think the code is simple enough and need no more explanation.

```
bool canCross(vector<int>& stones, int pos = 0, int k = 0) {
    for (int i = pos + 1; i < stones.size(); i++) {
        int gap = stones[i] - stones[pos];
        if (gap < k - 1) continue;
        if (gap > k + 1) return false;
        if (canCross(stones, i, gap)) return true;
    }
    return pos == stones.size() - 1;
}
```

This can pass OJ at 9ms but is inefficient for extreme cases. (update: new test cases are added and the solution above no longer passes OJ, please see the solution below which takes 62ms) We can memorize the returns with minimum effort:

```
unordered_map<int, bool> dp;

bool canCross(vector<int>& stones, int pos = 0, int k = 0) {
    int key = pos | k << 11;

    if (dp.count(key) > 0)
        return dp[key];

    for (int i = pos + 1; i < stones.size(); i++) {
        int gap = stones[i] - stones[pos];
        if (gap < k - 1)
            continue;
        if (gap > k + 1)
            return dp[key] = false;
        if (canCross(stones, i, gap))
            return dp[key] = true;
    }

    return dp[key] = (pos == stones.size() - 1);
}
```

The number of stones is less than 1100 so**pos** will always be less than 2^11 (2048). Stone positions could be theoretically up to 2^31 but **k** is practically not possible to be that big for the parameter as the steps must start from 0 and 1 and at the 1100th step the greatest valid k would be 1100. So combining **pos** and **k** is safe here.

written by mzchen original link here

Solution 2

For each stone, we write down a set of jump distances taken from previous stones to reach this stone (for example, if the stones were 3, 5, 7, then for stone 7 we write down 7 - 3 = 4 and 7 - 5 = 2, assuming they're both valid moves). From the distance set we can find a set of "imaginary" stones reachable from this stone, so all we have to do is to figure out (through hash table) which of those target stones actually exists, and propagate the distance to their distance sets.

Finally, we check if the last stone was reachable by checking if its distance set wasn't empty.

```
public class Solution {
    public boolean canCross(int[] stones) {
        final int l = stones != null ? stones.length : 0;
        if (l < 1 || stones[0] != 0) return false;
        final Map<Integer, Set<Integer>> map = new HashMap<>();
        for (int s : stones) map.put(s, new HashSet<Integer>());
        for (int s : stones) {
            Set<Integer> jSet = map.get(s);
            // Initial condition
            if (s == 0) {
                jSet.add(0);
                if (map.containsKey(1)) map.get(1).add(1);
                continue;
            }
            // For other stones
            for (int j : jSet) {
                int jj = j - 1;
                int ss = s + jj;
                // Previous jump - 1
                if (ss != s && map.containsKey(ss)) map.get(ss).add(jj);
                // Previous jump
                jj++; ss++;
                if (ss != s && map.containsKey(ss)) map.get(ss).add(jj);
                // Previous jump + 1
                jj++; ss++;
                if (ss != s && map.containsKey(ss)) map.get(ss).add(jj);
            }
        }
        return !map.get(stones[l - 1]).isEmpty();
    }
}
```

written by Rabby250 original link here

Solution 3

```
public boolean canCross(int[] stones) {
        if(stones[1] > 1) return false;
        if(stones.length == 2) return true;
        return helper(stones, 1, 1);
    }
    private boolean helper(int[] arr, int i, int step){
        boolean pass = false;
        if(i == arr.length-1) return true;
        for(int j = i+1; j < arr.length; j++){
            if(arr[j] <= arr[i] + step + 1 && arr[j] >= arr[i]+step-1){
                pass = pass || helper(arr, j, arr[j] - arr[i]);
            }
        }
        return pass;
    }
```

written by 类与对象Re original link here

From Leetcoder.

Sum of Left Leaves

Find the sum of all left leaves in a given binary tree.

Example:

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

```
There are two left leaves in the binary tree, with values 9 and 15 respectively. Re
turn 24.
```

Subscribe to see which companies asked this question

Solution 1

```
public class Solution {
    public int sumOfLeftLeaves(TreeNode root) {
        if(root == null || root.left == null && root.right == null) return 0;

        int res = 0;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        while(!queue.isEmpty()) {
            TreeNode curr = queue.poll();

            if(curr.left != null && curr.left.left == null && curr.left.right ==
null) res += curr.left.val;
            if(curr.left != null) queue.offer(curr.left);
            if(curr.right != null) queue.offer(curr.right);
        }
        return res;
    }
}
```

written by xietao0221 original link here

Solution 2

```
int sumOfLeftLeaves(TreeNode* root, bool isleft = false) {
    if (!root) return 0;
    if (!root->left && !root->right) return isleft ? root->val : 0;
    return sumOfLeftLeaves(root->left, true) + sumOfLeftLeaves(root->right, false
);
}
```

written by mzchen original link here

Solution 3

base case => node is none
recursive case => Left child is / isn't Leave

```
class Solution(object):
    def sumOfLeftLeaves(self, root):
        if not root: return 0
        if root.left and not root.left.left and not root.left.right:
            return root.left.val + self.sumOfLeftLeaves(root.right)
        return self.sumOfLeftLeaves(root.left) + self.sumOfLeftLeaves(root.right)
# isn't leave
```

EDIT:
Could be 3 Lines, but L2 would be too long.
thanks @tototo's advise!

written by YJL1228 original link here

From Leetcoder.

Convert a Number to Hexadecimal

Given an integer, write an algorithm to convert it to hexadecimal. For negative integer, two's complement method is used.

Note:

1. All letters in hexadecimal (`a-f`) must be in lowercase.
2. The hexadecimal string must not contain extra leading `0`s. If the number is zero, it is represented by a single zero character `'0'`; otherwise, the first character in the hexadecimal string will not be the zero character.
3. The given number is guaranteed to fit within the range of a 32-bit signed integer.
4. You **must not use *any* method provided by the library** which converts/formats the number to hex directly.

Example 1:

```
Input:
26

Output:
"1a"
```

Example 2:

```
Input:
-1

Output:
"ffffffff"
```

Subscribe to see which companies asked this question

Solution 1

```
/*
Basic idea: each time we take a look at the last four digits of
            binary verion of the input, and maps that to a hex char
            shift the input to the right by 4 bits, do it again
            until input becomes 0.

*/

public class Solution {

    char[] map = {'0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'}
;

    public String toHex(int num) {
        if(num == 0) return "0";
        String result = "";
        while(num != 0){
            result = map[(num & 15)] + result;
            num = (num >>> 4);
        }
        return result;
    }


}````
```

written by DonaldTrump original link here

Solution 2

```
  public String toHex(int dec) {
      if (dec == 0) return "0";
      StringBuilder res = new StringBuilder();

      while (dec != 0) {
          int digit = dec & 0xf;
          res.append(digit < 10 ? (char)(digit + '0') : (char)(digit - 10 + 'a'))
;
          dec >>>= 4;
      }

    return res.reverse().toString();
  }
```

written by yuhaowang001 original link here

Solution 3

```
public class Solution {
    public String toHex(int num) {
        return num == 0 ? "0" : toHex(num & 0xffffffffL);
    }

    public String toHex(long num) {
        return num < 16 ? hexdigit(num) : toHex(num / 16) + hexdigit(num % 16);
    }

    private String hexdigit(long num) {
        assert num < 16;
        return num < 10 ? Character.toString((char)(num + '0')) : Character.toStr
ing((char)(num - 10 + 'a'));
    }
}
```

written by dachuan.huang original link here

From Leetcoder.

Queue Reconstruction by Height

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers `(h, k)`, where `h` is the height of the person and `k` is the number of people in front of this person who have a height greater than or equal to `h`. Write an algorithm to reconstruct the queue.

Note:
The number of people is less than 1,100.

Example

```
Input:
[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:
[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]
```

Subscribe to see which companies asked this question

Solution 1

1. **Pick out tallest group of people and sort them** in a subarray (S). Since there's no other groups of people taller than them, therefore **each guy's index will be just as same as his k value**.
2. For 2nd tallest group (and the rest), insert each one of them into (S) by k value. So on and so forth.

E.g.
input: [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
subarray after step 1: **[[7,0], [7,1]]**
subarray after step 2: [[7,0], **[6,1]**, [7,1]]
...

It's not the most concise code, but I think it well explained the concept.

```
class Solution(object):
    def reconstructQueue(self, people):
        if not people: return []

        # obtain everyone's info
        # key=height, value=k-value, index in original array
        peopledct, height, res = {}, [], []

        for i in xrange(len(people)):
            p = people[i]
            if p[0] in peopledct:
                peopledct[p[0]] += (p[1], i),
            else:
                peopledct[p[0]] = [(p[1], i)]
                height += p[0],

        height.sort()      # here are different heights we have

        # sort from the tallest group
        for h in height[::-1]:
            peopledct[h].sort()
            for p in peopledct[h]:
                res.insert(p[0], people[p[1]])

        return res
```

EDIT:
Please also check:
@tlhuang 's concise Python code.
@tonygogogo 's 8 lines C++ solution.
@zeller2 's Java version.

written by YJL1228 original link here

Solution 2

```
vector<pair<int, int>> reconstructQueue(vector<pair<int, int>>& people) {
    auto comp = [](const pair<int, int>& p1, const pair<int, int>& p2)
                    { return p1.first > p2.first || (p1.first == p2.first && p1.s
econd < p2.second); };
    sort(people.begin(), people.end(), comp);
    vector<pair<int, int>> res;
    for (auto& p : people)
        res.insert(res.begin() + p.second, p);
    return res;
}
```

written by zyoppy008 original link here

Solution 3

Based on this solution which I first saw brought up by @bigoffer4all **here** (and which I **explained here**):

```
def reconstructQueue(self, people):
    queue = []
    for p in sorted(people, key=lambda (h, t): (-h, t)):
        queue.insert(p[1], p)
    return queue
```

That takes O(n^2) because each `insert` into the list takes O(n).

Instead of just one long list of all people, I break the queue into O(sqrt(n)) blocks of size up to sqrt(n). Then to insert at the desired index, I find the appropriate block, insert the person into that block, and potentially have to break the block into two. Each of those things takes O(sqrt(n)) time.

```
def reconstructQueue(self, people):
    blocks = [[]]
    for p in sorted(people, key=lambda (h, t): (-h, t)):
        index = p[1]

        for i, block in enumerate(blocks):
            m = len(block)
            if index <= m:
                break
            index -= m
        block.insert(index, p)
        if m * m > len(people):
            blocks.insert(i + 1, block[m/2:])
            del block[m/2:]

    return [p for block in blocks for p in block]
```

"Unfortunately", Python's `list.insert` is really fast compared to doing things in Python myself, and with the inputs allowed here (less than 1100 people), the O(n^2) solution wins. Locally I tested with larger inputs, and around 200000 people the two solutions were about equally fast. With 300000 people, the O(n sqrt(n)) solution was about factor 1.25 faster, and with a million people, the O(n sqrt(n)) solution was about factor 2.2 faster

The testing code:

```python
# The original O(n^2) solution.
class Solution(object):
    def reconstructQueue(self, people):
        queue = []
        for p in sorted(people, key=lambda (h, t): (-h, t)):
            queue.insert(p[1], p)
        return queue
nsquared = Solution().reconstructQueue

# The O(n sqrt(n)) solution.
class Solution(object):
    def reconstructQueue(self, people):
        blocks = [[]]
        for p in sorted(people, key=lambda (h, t): (-h, t)):
            index = p[1]

            for i, block in enumerate(blocks):
                m = len(block)
                if index <= m:
                    break
                index -= m
            block.insert(index, p)
            if m * m > len(people):
                blocks.insert(i + 1, block[m/2:])
                del block[m/2:]

        return [p for block in blocks for p in block]
nsqrtn = Solution().reconstructQueue

# Generate a large test case and time it.
from bisect import bisect
from random import randint, shuffle
from timeit import timeit
n = 300000
heights = [randint(1, n) for _ in range(n)]
standing = []
people = []
for h in heights:
    i = bisect(standing, -h)
    standing.insert(i, -h)
    people.append([h, i])
shuffle(people)
for solution in nsquared, nsqrtn, nsquared, nsqrtn:
    print timeit(lambda: solution(people), number=1)
```

written by StefanPochmann original link here

From Leetcoder.

Trapping Rain Water II

Given an `m x n` matrix of positive integers representing the height of each unit cell in a 2D elevation map, compute the volume of water it is able to trap after raining.

Note:
Both *m* and *n* are less than 110. The height of each unit cell is greater than 0 and is less than 20,000.

Example:

```
Given the following 3x6 height map:
[
  [1,4,3,1,3,2],
  [3,2,1,3,2,4],
  [2,3,3,2,3,1]
]

Return 4.
```

The above image represents the elevation map `[[1,4,3,1,3,2],[3,2,1,3,2,4],[2,3,3,2,3,1]]` before the rain.

After the rain, water are trapped between the blocks. The total volume of water trapped is 4.

Subscribe to see which companies asked this question

Solution 1

Source code from:
https://github.com/shawnfan/LintCode/blob/master/Java/Trapping Rain Water II.java

```
public class Solution {

    public class Cell {
        int row;
        int col;
        int height;
        public Cell(int row, int col, int height) {
            this.row = row;
            this.col = col;
            this.height = height;
        }
    }

    public int trapRainWater(int[][] heights) {
        if (heights == null || heights.length == 0 || heights[0].length == 0)
            return 0;

        PriorityQueue<Cell> queue = new PriorityQueue<>(1, new Comparator<Cell>()
{
            public int compare(Cell a, Cell b) {
                return a.height - b.height;
            }
        });

        int m = heights.length;
        int n = heights[0].length;
        boolean[][] visited = new boolean[m][n];

        // Initially, add all the Cells which are on borders to the queue.
        for (int i = 0; i < m; i++) {
            visited[i][0] = true;
            visited[i][n - 1] = true;
            queue.offer(new Cell(i, 0, heights[i][0]));
            queue.offer(new Cell(i, n - 1, heights[i][n - 1]));
        }

        for (int i = 0; i < n; i++) {
            visited[0][i] = true;
            visited[m - 1][i] = true;
            queue.offer(new Cell(0, i, heights[0][i]));
            queue.offer(new Cell(m - 1, i, heights[m - 1][i]));
        }

        // from the borders, pick the shortest cell visited and check its neighbo
rs:
        // if the neighbor is shorter, collect the water it can trap and update it
s height as its height plus the water trapped
       // add all its neighbors to the queue.
        int[][] dirs = new int[][]{{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
```

```
        int res = 0;
        while (!queue.isEmpty()) {
            Cell cell = queue.poll();
            for (int[] dir : dirs) {
                int row = cell.row + dir[0];
                int col = cell.col + dir[1];
                if (row >= 0 && row < m && col >= 0 && col < n && !visited[row][c
ol]) {
                    visited[row][col] = true;
                    res += Math.max(0, cell.height - heights[row][col]);
                    queue.offer(new Cell(row, col, Math.max(heights[row][col], ce
ll.height)));
                }
            }
        }

        return res;
    }
}
```

written by yuhaowang001 original link here

Solution 2

This problem can also be solved in a more general approach way using Dijkstra.

Construct a graph `G = (V, E)` as follows:
V = all cells plus a dummy vertex, `v`, corresponding to the outside region.
If `cell(i, j)` is adjacent to `cell(i', j')`, then add an direct edge from `(i, j)` to `(i', j')` with weight `height(i', j')`.
Add an edge with zero weight from any boundary cell to the dummy vertex `v`.

The weight of a path is defined as the weight of the heaviest edge along it. Then, for any cell `(i, j)`, the height of water it can save is equal to the weight, denoted by `dist(i, j)`, of the shortest path from `(i, j)` to `v`. (If the weight is less than or equal to `height(i, j)`, no water can be accumulated at that particular position.)

We want to compute the `dist(i, j)` for all pairs of `(i, j)`. Here, we have multiple sources and one destination, but this problem essentially can be solved using one pass of Dijkstra algorithm if we **reverse** the directions of all edges. The graph is sparse, i.e., there are O(rc) edges, resulting an `O(rc log(rc)) = O(rc max(log r, log c))` runtime and using `O(rc)` space.

Java Code:

```
public class Solution {

    int[] dx = {0, 0, 1, -1};
    int[] dy = {1, -1, 0, 0};

    List<int[]>[] g;
    int start;

    private int[] dijkstra() {
        int[] dist = new int[g.length];
        Arrays.fill(dist, Integer.MAX_VALUE / 2);
        dist[start] = 0;
        TreeSet<int[]> tree = new TreeSet<>((u, v) -> u[1] == v[1] ? u[0] - v[0]
: u[1] - v[1]);
        tree.add(new int[]{start, 0});
        while (!tree.isEmpty()) {
            int u = tree.first()[0], d = tree.pollFirst()[1];
            for (int[] e : g[u]) {
                int v = e[0], w = e[1];
                if (Math.max(d, w) < dist[v]) {
                    tree.remove(new int[]{v, dist[v]});
                    dist[v] = Math.max(d, w);
                    tree.add(new int[]{v, dist[v]});
                }
            }
        }
        return dist;
    }

    public int trapRainWater(int[][] a) {
        if (a == null || a.length == 0 || a[0].length == 0) return 0;
```

```
        int r = a.length, c = a[0].length;

        start = r * c;
        g = new List[r * c + 1];
        for (int i = 0; i < g.length; i++) g[i] = new ArrayList<>();
        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++) {
                if (i == 0 || i == r - 1 || j == 0 || j == c - 1) g[start].add(ne
w int[]{i * c + j, 0});
                for (int k = 0; k < 4; k++) {
                    int x = i + dx[k], y = j + dy[k];
                    if (x >= 0 && x < r && y >= 0 && y < c) g[i * c + j].add(new
int[]{x * c + y, a[i][j]});
                }
            }

        int ans = 0;
        int[] dist = dijkstra();
        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++) {
                int cb = dist[i * c + j];
                if (cb > a[i][j]) ans += cb - a[i][j];
            }

        return ans;
    }
}
```

written by lixx2100 original link here

Solution 3

```
/*
Basic physics:
Unlike bricks, water flows to wherever it could.
i.e we can't have the follwoing config made with water, but can do it with bricks
000
010
000
In the case above, if the "1" is built with water, that water can't stay. It needs
to be spilled!

2 steps Algorithm:
1. Since we know how to trap rain water in 1d, we can just transfor this 2D proble
m into 2 1D problems
    we go row by row, to calculate each spot's water
    we go column by column, to calculate each spot's water

2. Then, here comes the meat,
    For every spot that gets wet, from either row or column calculation, the wate
r can possibly spill.
    We need to check the water height aganist it's 4 neighbors.
        If the water height is taller than any one of its 4 neightbors, we need to
spill the extra water.
        If we spill any water from any slot, then its 4 neightbors needs to check
themselves again.
            For example, if we spill some water in the current slot b/c its bottm
neighbor's height, current slot's top neighbor's height might need to be updated
again.
        we keep checking until there is no water to be spilled.
*/


public class Solution {
    public int trapRainWater(int[][] heightMap) {
        /*FIRST STEP*/
        if(heightMap.length == 0) return 0;
        int[][] wetMap = new int[heightMap.length][heightMap[0].length];
        int sum = 0;
        /*row by row*/
        for(int i = 1; i < wetMap.length - 1; i++){
            wetMap[i] = calculate(heightMap[i]);
        }
        /*column by column*/
        for(int i = 1; i < heightMap[0].length - 1; i++){
            int[] col = new int[heightMap.length];
            for(int j = 0; j < heightMap.length; j++){
                col[j] = heightMap[j][i];
            }
            int[] colResult = calculate(col);
            /*update the wetMap to be the bigger value between row and col, later
we can spill, don't worry*/
            for(int j = 0; j < heightMap.length; j++){
                wetMap[j][i] = Math.max(colResult[j], wetMap[j][i]);
                sum += wetMap[j][i];
            }
```

```
        }
        /*SECOND STEP*/
        boolean spillWater = true;
        int[] rowOffset = {-1,1,0,0};
        int[] colOffset = {0,0,1,-1};
        while(spillWater){
            spillWater = false;
            for(int i = 1; i < heightMap.length - 1; i++){
                for(int j = 1; j < heightMap[0].length - 1; j++){
                    /*If this slot has ever gotten wet, exammine its 4 neightbors
*/
                    if(wetMap[i][j] != 0){
                        for(int m = 0; m < 4; m++){
                            int neighborRow = i + rowOffset[m];
                            int neighborCol = j + colOffset[m];
                            int currentHeight = wetMap[i][j] + heightMap[i][j];
                            int neighborHeight = wetMap[neighborRow][neighborCol]
+
                                                                    heightMap[neighbor
Row][neighborCol];
                            if(currentHeight > neighborHeight){
                                int spilledWater = currentHeight - Math.max(neigh
borHeight, heightMap[i][j]);
                                wetMap[i][j] = Math.max(0, wetMap[i][j] - spilled
Water);
                                sum -= spilledWater;
                                spillWater = true;
                            }
                        }
                    }
                }
            }
        }
        return sum;
    }

    /*Nothing interesting here, the same function for trapping water 1*/
    private int[] calculate (int[] height){
        int[] result = new int[height.length];
        Stack<Integer> s = new Stack<Integer>();
        int index = 0;
        while(index < height.length){
            if(s.isEmpty() || height[index] <= height[s.peek()]){
                s.push(index++);
            }else{
                int bottom = s.pop();
                if(s.size() != 0){
                    for(int i = s.peek() + 1; i < index; i++){
                        result[i] += (Math.min(height[s.peek()], height[index]) -
height[bottom]);
                    }
                }
            }
        }
        return result;
    }
```

```
}
```

written by DonaldTrump original link here

From Leetcoder.

Valid Word Abbreviation

Given a **non-empty** string `s` and an abbreviation `abbr`, return whether the string matches with the given abbreviation.

A string such as `"word"` contains only the following valid abbreviations:

```
["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1
", "1o2", "2r1", "3d", "w3", "4"]
```

Notice that only the above abbreviations are valid abbreviations of the string `"word"`. Any other string is not a valid abbreviation of `"word"`.

Note:
Assume `s` contains only lowercase letters and `abbr` contains only lowercase letters and digits.

Example 1:

```
Given s = "internationalization", abbr = "i12iz4n":

Return true.
```

Example 2:

```
Given s = "apple", abbr = "a2e":

Return false.
```

Solution 1

```
    public boolean validWordAbbreviation(String word, String abbr) {
        int i = 0, j = 0;
        while (i < word.length() && j < abbr.length()) {
            if (word.charAt(i) == abbr.charAt(j)) {
                ++i;++j;
                continue;
            }
            if (abbr.charAt(j) <= '0' || abbr.charAt(j) > '9') {
                return false;
            }
            int start = j;
            while (j < abbr.length() && abbr.charAt(j) >= '0' && abbr.charAt(j) <
= '9') {
                ++j;
            }
            int num = Integer.valueOf(abbr.substring(start, j));
            i += num;
        }
        return i == word.length() && j == abbr.length();
    }
```

written by lzmshiwo original link here

Solution 2

Update

Much nicer, I just turn an abbreviation like `"i12iz4n"` into a regular expression like `"i.{12}iz.{4}n"`. Duh.

Java:

```
public boolean validWordAbbreviation(String word, String abbr) {
    return word.matches(abbr.replaceAll("[1-9]\\d*", ".{$0}"));
}
```

Python:

```
def validWordAbbreviation(self, word, abbr):
    return bool(re.match(re.sub('([1-9]\d*)', r'.{\1}', abbr) + '$', word))
```

Obsolete original

(This now gets a memory error, since the exploding testcase I suggested at the end has been added to the test suite.)

"Clean":

```
def validWordAbbreviation(self, word, abbr):
    regex = re.sub('\d+', lambda m: '.' * int(m.group()), abbr)
    return bool(re.match(regex + '$', word)) and not re.search('(^|\D)0', abbr)
```

"Dirty" (abusing how Python handles the `>` there):

```
def validWordAbbreviation(self, word, abbr):
    regex = re.sub('\d+', lambda m: '.' * int(m.group()), abbr)
    return re.match(regex + '$', word) > re.search('(^|\D)0', abbr)
```

I turn each number into that many dots to get a regular expression. For example, when asked whether `"3t2de"` is a valid abbreviation for word `"leetcode"`, I turn `"3t2de"` into `"...t..de"` and check whether that regular expression matches `"leetcode"`, which it does. I also need to rule out the number `"0"` and leading zeros, which I do with another regular expression.

@1337c0d3r I suggest adding test case `"bignumberhahaha", "999999999"`, as that gets me a fully deserved MemoryError :-)

written by StefanPochmann original link here

Solution 3

```
class Solution {
public:
    bool validWordAbbreviation(string word, string abbr) {
        int i = 0, j = 0;
        while (i < word.size() && j < abbr.size()) {
            if (isdigit(abbr[j])) {
                if (abbr[j] == '0') return false;
                int val = 0;
                while (j < abbr.size() && isdigit(abbr[j]))
                    val = val * 10 + abbr[j++] - '0';
                i += val;
            }
            else if (word[i++] != abbr[j++]) return false;
        }
        return i == word.size() && j == abbr.size();
    }
};
```

written by zyoppy008 original link here

From Leetcoder.

Longest Palindrome

Given a string which consists of lowercase or uppercase letters, find the length of the longest palindromes that can be built with those letters.

This is case sensitive, for example `"Aa"` is not considered a palindrome here.

Note:
Assume the length of given string will not exceed 1,010.

Example:

```
Input:
"abcccdd"

Output:
7

Explanation:
One longest palindrome that can be built is "dccaccd", whose length is 7.
```

Solution 1

```
public int longestPalindrome(String s) {
        if(s==null || s.length()==0) return 0;
        HashSet<Character> hs = new HashSet<Character>();
        int count = 0;
        for(int i=0; i<s.length(); i++){
            if(hs.contains(s.charAt(i))){
                hs.remove(s.charAt(i));
                count++;
            }else{
                hs.add(s.charAt(i));
            }
        }
        if(!hs.isEmpty()) return count*2+1;
        return count*2;
}
```

written by charliebob0604gmail.com original link here

Solution 2

I count how many letters appear an odd number of times. Because we can use**all** letters, except for each odd-count letter we must leave one, except one of them we can use.

Python:

```
def longestPalindrome(self, s):
    odds = sum(v & 1 for v in collections.Counter(s).values())
    return len(s) - odds + bool(odds)
```

C++:

```
int longestPalindrome(string s) {
    int odds = 0;
    for (char c='A'; c<='z'; c++)
        odds += count(s.begin(), s.end(), c) & 1;
    return s.size() - odds + (odds > 0);
}
```

Similar solutions (I actually like the `use` solutions better than the above, but I'm just so fond of my topic title :-)

```
def longestPalindrome(self, s):
    use = sum(v & ~1 for v in collections.Counter(s).values())
    return use + (use < len(s))

def longestPalindrome(self, s):
    counts = collections.Counter(s).values()
    return sum(v & ~1 for v in counts) + any(v & 1 for v in counts)

int longestPalindrome(string s) {
    int use = 0;
    for (char c='A'; c<='z'; c++)
        use += count(s.begin(), s.end(), c) & ~1;
    return use + (use < s.size());
}

int longestPalindrome(string s) {
    vector<int> count(256);
    for (char c : s)
        ++count[c];
    int odds = 0;
    for (int c : count)
        odds += c & 1;
    return s.size() - odds + (odds > 0);
}

int longestPalindrome(string s) {
    vector<int> count(256);
    int odds = 0;
    for (char c : s)
        odds += ++count[c] & 1 ? 1 : -1;
    return s.size() - odds + (odds > 0);
}
```

written by StefanPochmann original link here

Solution 3

```
class Solution {
public:
    int longestPalindrome(string s) {
        vector<int> m(256, 0);
        for (auto& c : s) m[c-'\0']++;
        int result = 0;
        for (auto& i : m) result += i%2 ? (result%2 ? i-1 : i) : i;
        return result;
    }
};
```

written by vesion original link here

From Leetcoder.

Split Array Largest Sum

Given an array which consists of non-negative integers and an integer *m*, you can split the array into *m* non-empty continuous subarrays. Write an algorithm to minimize the largest sum among these *m* subarrays.

Note:
Given *m* satisfies the following constraint: 1 ≤ m ≤ length(nums) ≤ 14,000.

Examples:

```
Input:
nums = [1,2,3,4,5]
m = 2

Output:
9

Explanation:
There are four ways to split nums into two subarrays.
The best way is to split it into [1,2,3] and [4,5],
where the largest sum among the two subarrays is only 9.
```

Solution 1

1. The answer is between maximum value of input array numbers and sum of those numbers.
2. Use binary search to approach the correct answer. We have `l = max number of array; r = sum of all numbers in the array;` Every time we do `mid = (l + r) / 2;`
3. Use greedy to narrow down left and right boundaries in binary search.
   3.1 Cut the array from left.
   3.2 Try our best to make sure that the sum of numbers between each two cuts (inclusive) is large enough but still less than `mid`.
   3.3 We'll end up with two results: either we can divide the array into more than m subarrays or we cannot.
   If we can, it means that the `mid` value we pick is too small because we've already tried our best to make sure each part holds as many non-negative numbers as we can but we still have numbers left. So, it is impossible to cut the array into m parts and make sure each parts is no larger than `mid`. We should increase m. This leads to `l = mid + 1;`
   If we can't, it is either we successfully divide the array into m parts and the sum of each part is less than `mid`, or we used up all numbers before we reach m. Both of them mean that we should lower `mid` because we need to find the minimum one. This leads to `r = mid - 1;`

```java
public class Solution {
    public int splitArray(int[] nums, int m) {
        int max = 0; long sum = 0;
        for (int num : nums) {
            max = Math.max(num, max);
            sum += num;
        }
        if (m == 1) return (int)sum;
        //binary search
        long l = max; long r = sum;
        while (l <= r) {
            long mid = (l + r)/ 2;
            if (valid(mid, nums, m)) {
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
        return (int)l;
    }
    public boolean valid(long target, int[] nums, int m) {
        int count = 1;
        long total = 0;
        for(int num : nums) {
            total += num;
            if (total > target) {
                total = num;
                count++;
                if (count > m) {
                    return false;
                }
            }
        }
        return true;
    }
}
```

- list item

written by dax1ng original link here

Solution 2

1. Given a result, it is easy to test whether it is valid or not.
2. The max of the result is the sum of the input nums.
3. The min of the result is the max num of the input nums.
   Given the 3 conditions above we can do a binary search. (need to deal with overflow)

```
public class Solution {
    public int splitArray(int[] nums, int m) {
        long sum = 0;
        int max = 0;
        for(int num: nums){
            max = Math.max(max, num);
            sum += num;
        }
        return (int)binary(nums, m, sum, max);
    }

    private long binary(int[] nums, int m, long high, long low){
        long mid = 0;
        while(low < high){
            mid = (high + low)/2;
            if(valid(nums, m, mid)){
                //System.out.println(mid);
                high = mid;
            }else{
                low = mid + 1;
            }
        }
        return high;
    }

    private boolean valid(int[] nums, int m, long max){
        int cur = 0;
        int count = 1;
        for(int num: nums){
            cur += num;
            if(cur > max){
                cur = num;
                count++;
                if(count > m){
                    return false;
                }
            }
        }
        return true;
    }
}
```

written by zrythpzhl original link here

Solution 3

Obviously, the final result is in the interval [left, right] (where left is the maximal number in the array, right is sum of all numbers).
So, what we need to do is to find out the first element in [left, right], which exactly we cannot split the array into m subarrays whose sum is no greater than that element. Then its previous one is the final result. The progress is much similar to lower_bound in C++.

```
class Solution {
public:
    using ll = long long;

    bool canSplit(vector<int>& nums, int m, ll sum) {
        int c = 1;
        ll s = 0;
        for (auto& num : nums) {
            s += num;
            if (s > sum) {
                s = num;
                ++c;
            }
        }
        return c <= m;
    }

    int splitArray(vector<int>& nums, int m) {
        ll left = 0, right = 0;
        for (auto& num : nums) {
            left = max(left, (ll)num);
            right += num;
        }
        while (left <= right) {
            ll mid = left + (right-left)/2;
            if (canSplit(nums, m, mid))
                right = mid-1;
            else
                left = mid+1;
        }
        return left;
    }
};
```

written by vesion original link here

From Leetcoder.

Minimum Unique Word Abbreviation

A string such as `"word"` contains the following abbreviations:

```
["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1
", "1o2", "2r1", "3d", "w3", "4"]
```

Given a target string and a set of strings in a dictionary, find an abbreviation of this target string with the ***smallest possible*** length such that it does not conflict with abbreviations of the strings in the dictionary.

Each **number** or letter in the abbreviation is considered length = 1. For example, the abbreviation "a32bc" has length = 4.

Note:

- In the case of multiple answers as shown in the second example below, you may return any one of them.
- Assume length of target string = **m**, and dictionary size = **n**. You may assume that $\mathbf{m \le 21}$, $\mathbf{n \le 1000}$, and $\mathbf{\log_2(n) + m \le 20}$.

Examples:

```
"apple", ["blade"] -> "a4" (because "5" or "4e" conflicts with "blade")

"apple", ["plain", "amber", "blade"] -> "1p3" (other valid answers include "ap3", "
a3e", "2p2", "3le", "3l1").
```

Solution 1

I think this problem is NP-hard. For proof, I'll reduce the **set cover** decision problem to it. The example there is:

```
U = {1, 2, 3, 4, 5}
S = {{1, 2, 3}, {2, 4}, {3, 4}, {4, 5}}
```

Now I'd encode that for **Minimum Unique Word Abbreviation** like this:

```
target: "oooo"
dict:   "looo" (word 1)
        "lloo" (word 2)
        "lolo" (word 3)
        "olll" (word 4)
        "oool" (word 5)
```

The five dictionary words correspond to the five elements of U. And every word has four letters, corresponding to the four subsets in S. You can see for example the second subset {2, 4} encoded as the second colum in the dictionary, which has `l` in words 2 and 4 (and otherwise only `o`).

(Side note: You can build an abbreviation by picking letters and then replacing the unpicked ones with numbers. For example, when you have `leetcode` and pick the letters `l`, `t` and the last `e`, you get `l2t3e`. This way you can get all $2^{|\text{word}|}$ possible abbreviations from the $2^{|\text{word}|}$ possible ways of picking, as I've done in the first solution here.)

Now an optimal abbreviation for the target is "o2o", which we get by picking the first "o" because that distinguishes the target from words {1, 2, 3} and by picking the last "o" because that distinguishes the target from words {4, 5}. Just like picking the subsets {1, 2, 3} and {4, 5} cover all of U in the set cover problem.

A little problem is that if I switch the third and fourth element of S, then an optimal abbreviation picks the first and *third* "o", so instead of "o2o" we'd get "o1o1". That's a different length. So instead of the above encoding, I'd really do this, inserting "x" before and after each letter:

```
target: "xoxoxoxox"
dict:   "xlxoxoxox"
        "xlxlxoxox"
        "xlxoxlxox"
        "xoxlxlxlx"
        "xoxoxoxlx"
```

Now "1o5o1" is an optimal abbreviation, and when I switch third and fourth element of S again, I instead get "1o3o3", which still has the same length.

To be precise and usable: When I pick k letters, the optimal abbreviation will have those k letters as well as k+1 numbers, so length 2k+1. That means minimizing the

abbreviation length is equivalent to minimizing the number of picked letters, which is equivalent to minimizing the number of picked elements of S in the decision problem.

Summary: To solve a given **set cover** decision problem instance, I translate it to a **Minimum Unique Word Abbreviation** instance like above, and then from the computed optimal abbreviation, I subtract 1 from its length and divide it by 2 to get the number of picked letters, which is the minimum number of elements of S that cover U. And with that, I can directly answer the original set cover instance.

Moral of the story: Don't feel bad about writing a brute force solution :-)

written by StefanPochmann original link here

Solution 2

The key idea of my solution is to preprocess the dictionary to transfer all the words to bit sequences (int):
Pick the words with same length as target string from the dictionary and compare the characters with target. If the characters are different, set the corresponding bit to 1, otherwise, set to 0.
Ex: "abcde", ["abxdx", "xbcdx"] => [00101, 10001]

The problem is now converted to find a bit mask that can represent the shortest abbreviation, so that for all the bit sequences in dictionary, mask & bit sequence > 0.
Ex: for [00101, 10001], the mask should be [00001]. if we mask the target string with it, we get "****e" ("4e"), which is the abbreviation we are looking for.

To find the bit mask, we need to perform DFS with some optimizations. But which bits should be checked? We can perform "or" operation for all the bit sequences in the dictionary and do DFS for the "1" bits in the result.
Ex: 00101 | 10001 = 10101, so we only need to take care of the 1st, 3rd, and 5th bit.

Here is a C++ implementation, the running time should be about 3ms. Any suggestions would be appreciated.

```
class Solution {
    int n, cand, bn, minlen, minab;
    vector<int> dict;

    // Return the length of abbreviation given bit sequence
    int abbrLen(int mask) {
        int count = 0;
        for (int b = 1; b < bn;) {
            if ((mask & b) == 0)
                for (; b < bn and (mask & b) == 0; b <<= 1);
            else b <<= 1;
            count ++;
        }
        return count;
    }

    // DFS backtracking
    void dfs(int bit, int mask) {
        int len = abbrLen(mask);
        if (len >= minlen) return;
        bool match = true;
        for (auto d : dict) {
            if ((mask & d) == 0) {
                match = false;
                break;
            }
        }
        if (match) {
            minlen = len;
            minab = mask;
        }
        else
            for (int b = bit; b < bn; b <<= 1)
```

```
            for (int b = bit; b < bn; b <<= 1)
                if (cand & b) dfs(b << 1, mask + b);
    }

public:
    string minAbbreviation(string target, vector<string>& dictionary) {
        n = target.size(), bn = 1 << n, cand = 0, minlen = INT_MAX;
        string res;

        // Preprocessing with bit manipulation
        for (auto w : dictionary) {
            int word = 0;
            if (w.size() != n) continue;
            for (int i = n-1, bit = 1; i >= 0; --i, bit <<= 1)
                if (target[i] != w[i]) word += bit;
            dict.push_back(word);
            cand |= word;
        }
        dfs(1, 0);

        // Reconstruct abbreviation from bit sequence
        for (int i = n-1, pre = i; i >= 0; --i, minab >>= 1) {
            if (minab & 1) {
                if (pre-i > 0) res = to_string(pre-i) + res;
                pre = i - 1;
                res = target[i] + res;
            }
            else if (i == 0) res = to_string(pre-i+1) + res;
        }
        return res;
    }
};
```

UPDATE: a better way to determine the length of abbreviation mentioned by @StefanPochmann

```
int abbrLen(int mask) {
    int count = n;
    for (int b = 3; b < bn; b <<= 1)
        if ((mask & b) == 0)
            count --;
    return count;
}
```

written by topcoder007 original link here

Solution 3

Gets accepted in ~130 ms.

```
def minAbbreviation(self, target, dictionary):
    m = len(target)
    diffs = {sum(2**i for i, c in enumerate(word) if target[i] != c)
             for word in dictionary if len(word) == m}
    if not diffs:
        return str(m)
    bits = max((i for i in range(2**m) if all(d & i for d in diffs)),
               key=lambda bits: sum((bits >> i) & 3 == 0 for i in range(m-1)))
    s = ''.join(target[i] if bits & 2**i else '#' for i in range(m))
    return re.sub('#+', lambda m: str(len(m.group())), s)
```

If the target is `apple` and the dictionary contains `apply`, then the abbreviation must include the `e` as the letter `e`, not in a number. It's the only letter distinguishing these two words. Similarly, if the dictionary contains `tuple`, then the abbreviation must include the `a` or the first `p` as a letter.

For each dictionary word (of correct size), I create a diff-number whose bits tell me which of the word's letters differ from the target. Then I go through the 2^m possible abbreviations, represented as number from 0 to 2^m-1, the bits representing which letters of target are in the abbreviation. An abbreviation is ok if it doesn't match any dictionary word. To check whether an abbreviation doesn't match a dictionary word, I simply check whether the abbreviation number and the dictionary word's diff-number have a common 1-bit. Which means that the abbreviation contains a letter where the dictionary word differs from the target.

Then from the ok abbreviations I find one that maximizes how much length it saves me. Two consecutive 0-bits in the abbreviation number mean that the two corresponding letters will be encoded as the number 2. It saves length 1. Three consecutive 0-bits save length 2, and so on. To compute the saved length, I just count how many pairs of adjacent bits are zero.

Now that I have the number representing an optimal abbreviation, I just need to turn it into the actual abbreviation. First I turn it into a string where each 1-bit is turned into the corresponding letter of the target and each 0-bit is turned into `#`. Then I replace streaks of `#` into numbers.

written by StefanPochmann original link here

From Leetcoder.

Add Strings

Given two non-negative numbers `num1` and `num2` represented as string, return the sum of `num1` and `num2`.

Note:

1. The length of both `num1` and `num2` is
2. Both `num1` and `num2` contains only digits `0-9`.
3. Both `num1` and `num2` does not contain any leading zero.
4. You **must not use any built-in BigInteger library** or **convert the inputs to integer** directly.

Solution 1

```
class Solution {
public:
string addStrings(string num1, string num2) {
    int n1 = num1.size(), i = n1 - 1;
    int n2 = num2.size(), j = n2 - 1;
    int carry = 0;
    string res = "";
    while(i>=0 || j>=0 || carry){
        long sum = 0;
        if(i >= 0){sum += (num1[i] - '0');i--;}
        if(j >= 0){sum += (num2[j] - '0');j--;}
        sum += carry;
        carry = sum / 10;
        sum = sum % 10;
        res = to_string(sum) + res;
    }
    return res;
}
};
```

written by jasonshieh original link here

Solution 2

```
public class Solution {
    public String addStrings(String num1, String num2) {
        StringBuilder sb = new StringBuilder();
        int carry = 0;
        for(int i = num1.length() - 1, j = num2.length() - 1; i >= 0 || j >= 0; i
--, j--){
            int x = i < 0 ? 0 : num1.charAt(i) - '0';
            int y = j < 0 ? 0 : num2.charAt(j) - '0';
            sb.append((x + y + carry) % 10);
            carry = (x + y + carry) / 10;
        }
        if(carry != 0)
            sb.append(carry);
        return sb.reverse().toString();
    }
}
```

written by ZachC original link here

Solution 3

```
public String addStrings(String num1, String num2) {
        int len1=num1.length()-1;
        int len2=num2.length()-1;

        StringBuilder sb=new StringBuilder();
        int sum=0,carry=0;
        while(len1>=0 || len2>=0) {
            int first=len1>=0?num1.charAt(len1)-'0':0;
            int second=len2>=0?num2.charAt(len2)-'0':0;
            sum=carry+first+second;
            if(sum<=9){
                sb.insert(0,sum);
                sum=0;
                carry=0;
            } else {
                sb.insert(0,sum%10);
                sum=0;
                carry=1;
            }
            len1--;
            len2--;
        }
    if(carry==1)sb.insert(0,"1");
    return sb.toString();
    }
```

written by guruvignesh original link here

From Leetcoder.

Partition Equal Subset Sum

Given a **non-empty** array containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Note:

1. Each of the array element will not exceed 100.
2. The array size will not exceed 200.

Example 1:

```
Input: [1, 5, 11, 5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].
```

Example 2:

```
Input: [1, 2, 3, 5]

Output: false

Explanation: The array cannot be partitioned into equal sum subsets.
```

Solution 1

```java
public class Solution {
    public boolean canPartition(int[] nums) {
        // check edge case
        if (nums == null || nums.length == 0) {
            return true;
        }
        // preprocess
        int volumn = 0;
        for (int num : nums) {
            volumn += num;
        }
        if (volumn % 2 != 0) {
            return false;
        }
        volumn /= 2;
        // dp def
        boolean[] dp = new boolean[volumn + 1];
        // dp init
        dp[0] = true;
        // dp transition
        for (int i = 1; i <= nums.length; i++) {
            for (int j = volumn; j >= nums[i-1]; j--) {
                dp[j] = dp[j] || dp[j - nums[i-1]];
            }
        }
        return dp[volumn];
    }
}
```

written by tao62 original link here

Solution 2

1. DFS solution:

```
class Solution {
public:
    bool backtrack(vector<int>& nums, int start, int target) {
        if (target <= 0) return target == 0;
        for (int i = start; i < nums.size(); i++)
            if (backtrack(nums, i + 1, target - nums[i])) return true;
        return false;
    }

    bool canPartition(vector<int>& nums) {
        int sum = accumulate(nums.begin(), nums.end(), 0);
        return !(sum & 1) && backtrack(nums, 0, sum >> 1);
    }
};
```

2. DFS can't pass the OJ, as more test cases are added. So here comes a DP solution based on @Hermits solution

```
bool canPartition(vector<int>& nums) {
    int sum = accumulate(nums.begin(), nums.end(), 0), target = sum >> 1;
    if (sum & 1) return false;
    vector<int> dp(target + 1, 0);
    dp[0] = 1;
    for(auto num : nums)
        for(int i = target; i >= num; i--)
            dp[i] = dp[i] || dp[i - num];
    return dp[target];
}
```

3. A very fast and cool Bit solution by @alvin-777 solution

```
bool canPartition(vector<int>& nums) {
    bitset<5001> bits(1);
    int sum = accumulate(nums.begin(), nums.end(), 0);
    for (auto n : nums) bits |= bits << n;
    return !(sum & 1) && bits[sum >> 1];
}
```

written by zyoppy008 original link here

Solution 3

Time complexity O(n), size of the bitset is 632 bytes

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        bitset<5001> bits(1);
        int sum = 0;
        for (auto n : nums) {
            sum += n;
            bits |= bits << n;
        }
        return !(sum & 1) && bits[sum >> 1]; // !(sum % 2) && bits[sum / 2];
    }
};
```

It's possible to shorten the solution to 4 lines, by using std::accumulate(), but that doesn't really make you type less or make it run faster though...

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        bitset<5001> bits(1);
        int sum = accumulate(nums.begin(), nums.end(), 0);
        for (auto n : nums) bits |= bits << n;
        return !(sum & 1) && bits[sum >> 1];
    }
};
```

written by alvin-777 original link here

From Leetcoder.

Pacific Atlantic Water Flow

Given an `m x n` matrix of non-negative integers representing the height of each unit cell in a continent, the "Pacific ocean" touches the left and top edges of the matrix and the "Atlantic ocean" touches the right and bottom edges.

Water can only flow in four directions (up, down, left, or right) from a cell to another one with height equal or lower.

Find the list of grid coordinates where water can flow to both the Pacific and Atlantic ocean.

Note:

1. The order of returned grid coordinates does not matter.
2. Both *m* and *n* are less than 150.

Example:

```
Given the following 5x5 matrix:

  Pacific ~   ~   ~   ~   ~
       ~  1   2   2   3  (5) *
       ~  3   2   3  (4) (4) *
       ~  2   4  (5)  3   1  *
       ~ (6) (7)  1   4   5  *
       ~ (5)  1   1   2   4  *
          *   *   *   *   * Atlantic

Return:

[[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]] (positions with parenthese
s in above matrix).
```

Solution 1

1. Two Queue and add all the Pacific border to one queue; Atlantic border to another queue.
2. Keep a visited matrix for each queue. In the end, add the cell visited by two queue to the result.
   BFS: Water flood from ocean to the cell. Since water can only flow from high/equal cell to low cell, add the neighboor cell with height larger or equal to current cell to the queue and mark as visited.

```java
public class Solution {
    int[][]dir = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    public List<int[]> pacificAtlantic(int[][] matrix) {
        List<int[]> res = new LinkedList<>();
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0){
            return res;
        }
        int n = matrix.length, m = matrix[0].length;
        //One visited map for each ocean
        boolean[][] pacific = new boolean[n][m];
        boolean[][] atlantic = new boolean[n][m];
        Queue<int[]> pQueue = new LinkedList<>();
        Queue<int[]> aQueue = new LinkedList<>();
        for(int i=0; i<n; i++){ //Vertical border
            pQueue.offer(new int[]{i, 0});
            aQueue.offer(new int[]{i, m-1});
            pacific[i][0] = true;
            atlantic[i][m-1] = true;
        }
        for(int i=0; i<m; i++){ //Horizontal border
            pQueue.offer(new int[]{0, i});
            aQueue.offer(new int[]{n-1, i});
            pacific[0][i] = true;
            atlantic[n-1][i] = true;
        }
        bfs(matrix, pQueue, pacific);
        bfs(matrix, aQueue, atlantic);
        for(int i=0; i<n; i++){
            for(int j=0; j<m; j++){
                if(pacific[i][j] && atlantic[i][j])
                    res.add(new int[]{i,j});
            }
        }
        return res;
    }
    public void bfs(int[][]matrix, Queue<int[]> queue, boolean[][]visited){
        int n = matrix.length, m = matrix[0].length;
        while(!queue.isEmpty()){
            int[] cur = queue.poll();
            for(int[] d:dir){
                int x = cur[0]+d[0];
                int y = cur[1]+d[1];
                if(x<0 || x>=n || y<0 || y>=m || visited[x][y] || matrix[x][y] <
matrix[cur[0]][cur[1]]){
                    continue;
                }
                visited[x][y] = true;
                queue.offer(new int[]{x, y});
            }
        }
    }
}
```

DFS version:

```java
public class Solution {
    public List<int[]> pacificAtlantic(int[][] matrix) {
        List<int[]> res = new LinkedList<>();
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0){
            return res;
        }
        int n = matrix.length, m = matrix[0].length;
        boolean[][]pacific = new boolean[n][m];
        boolean[][]atlantic = new boolean[n][m];
        for(int i=0; i<n; i++){
            dfs(matrix, pacific, Integer.MIN_VALUE, i, 0);
            dfs(matrix, atlantic, Integer.MIN_VALUE, i, m-1);
        }
        for(int i=0; i<m; i++){
            dfs(matrix, pacific, Integer.MIN_VALUE, 0, i);
            dfs(matrix, atlantic, Integer.MIN_VALUE, n-1, i);
        }
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                if (pacific[i][j] && atlantic[i][j])
                    res.add(new int[] {i, j});
        return res;
    }

    int[][]dir = new int[][]{{0,1},{0,-1},{1,0},{-1,0}};

    public void dfs(int[][]matrix, boolean[][]visited, int height, int x, int y){
        int n = matrix.length, m = matrix[0].length;
        if(x<0 || x>=n || y<0 || y>=m || visited[x][y] || matrix[x][y] < height)
            return;
        visited[x][y] = true;
        for(int[]d:dir){
            dfs(matrix, visited, matrix[x][y], x+d[0], y+d[1]);
        }
    }
}
```

written by star1993 original link here

Solution 2

```
class Solution {
public:
    vector<pair<int, int>> res;
    vector<vector<int>> visited;
    void dfs(vector<vector<int>>& matrix, int x, int y, int pre, int preval){
        if (x < 0 || x >= matrix.size() || y < 0 || y >= matrix[0].size()
                || matrix[x][y] < pre || (visited[x][y] & preval) == preval)
            return;
        visited[x][y] |= preval;
        if (visited[x][y] == 3) res.push_back({x, y});
        dfs(matrix, x + 1, y, matrix[x][y], visited[x][y]); dfs(matrix, x - 1, y,
matrix[x][y], visited[x][y]);
        dfs(matrix, x, y + 1, matrix[x][y], visited[x][y]); dfs(matrix, x, y - 1,
matrix[x][y], visited[x][y]);
    }

    vector<pair<int, int>> pacificAtlantic(vector<vector<int>>& matrix) {
        if (matrix.empty()) return res;
        int m = matrix.size(), n = matrix[0].size();
        visited.resize(m, vector<int>(n, 0));
        for (int i = 0; i < m; i++) {
            dfs(matrix, i, 0, INT_MIN, 1);
            dfs(matrix, i, n - 1, INT_MIN, 2);
        }
        for (int i = 0; i < n; i++) {
            dfs(matrix, 0, i, INT_MIN, 1);
            dfs(matrix, m - 1, i, INT_MIN, 2);
        }
        return res;
    }
};
```

written by zyoppy008 original link here

Solution 3

```
/*
1.Naive solution:
    Standard dfs, which means for each point, we check if it can reach both pacifi
c and atlantic,
    for each point, we can possibly check all the rest of points, O(m*n * m*n)

2.A little improvement:
    What about we 4 hash tables, they keep track of all the points we know so far
that
        can reach atlantic
        cannot reach atlantic
        can reach pacific
        cannot reach pacific
    It's doable, still hit TLE, although I didn't hit TLE when not submitting the
code, but running it using the provided testing environment

3.On the other hand, we can consider the flip side
    We can let the pacific and atlantic ocean "flow into" the matrix as much as po
ssible,
    using 2 boolean arrays, one for each ocean.
    The result are the points that are true in both boolean table
*/


public class Solution {
    public List<int[]> pacificAtlantic(int[][] matrix) {
        List<int[]> result = new ArrayList<int[]>();
        if(matrix.length == 0 || matrix[0].length == 0) return result;
        boolean[][] pacific = new boolean[matrix.length][matrix[0].length];  // t
he pacific boolean table
        boolean[][] atlantic = new boolean[matrix.length][matrix[0].length]; // t
he atlantic booean table
        //initially, all the top and left cells are flooded with pacific water
        //and all the right and bottom cells are flooded with atlantic water
        for(int i = 0; i < matrix.length; i++){
            pacific[i][0] = true;
            atlantic[i][matrix[0].length-1] = true;
        }
        for(int i = 0; i < matrix[0].length; i++){
            pacific[0][i] = true;
            atlantic[matrix.length-1][i] = true;
        }
        //we go around the matrix and try to flood the matrix from 4 side.
        for(int i = 0; i < matrix.length; i++){
            boolean[][] pacificVisited = new boolean[matrix.length][matrix[0].len
gth];
            boolean[][] atlanticVisited = new boolean[matrix.length][matrix[0].le
ngth];
            water(pacific, pacificVisited, matrix, i,0);
            water(atlantic, atlanticVisited, matrix, i, matrix[0].length - 1);

        }
        for(int i = 0; i < matrix[0].length; i++){
            boolean[][] pacificVisited = new boolean[matrix.length][matrix[0].len
```

```
gth];
            boolean[][] atlanticVisited = new boolean[matrix.length][matrix[0].le
ngth];
            water(pacific, pacificVisited, matrix, 0,i);
            water(atlantic, atlanticVisited, matrix, matrix.length - 1, i);

        }
        //check the shared points among 2 tables
        for(int i = 0; i < matrix.length; i++){
            for(int j = 0; j < matrix[0].length; j++){
                if(pacific[i][j] && atlantic[i][j]){
                    int[] element = {i,j};
                    result.add(element);
                }
            }
        }
        return result;
    }
    //the flood function
    private void water(boolean[][] wet, boolean[][] visited, int[][] matrix, int
i , int j){
        wet[i][j] = true;
        visited[i][j] = true;
        int[] x = {0,0,1,-1};
        int[] y = {1,-1,0,0};
        for(int k = 0; k < 4; k++){
            if(i+y[k] >= 0 && i+y[k] < matrix.length && j+x[k] >= 0 && j+x[k] < m
atrix[0].length
                && !visited[i+y[k]][j+x[k]] && matrix[i+y[k]][j+x[k]] >= matrix[i
][j]){
                water(wet, visited, matrix, i+y[k], j+x[k]);
            }
        }
    }
}```

P.S Sometimes you choose an option just because the alternative is just worse.....
```

written by DonaldTrump original link here

From Leetcoder.

Sentence Screen Fitting

Given a `rows x cols` screen and a sentence represented by a list of words, find **how many times** the given sentence can be fitted on the screen.

Note:

1. A word cannot be split into two lines.
2. The order of words in the sentence must remain unchanged.
3. Two consecutive words **in a line** must be separated by a single space.
4. Total words in the sentence won't exceed 100.
5. Length of each word won't exceed 10.
6. 1 ≤ rows, cols ≤ 20,000.

Example 1:

```
Input:
rows = 2, cols = 8, sentence = ["hello", "world"]

Output:
1

Explanation:
hello---
world---

The character '-' signifies an empty space on the screen.
```

Example 2:

```
Input:
rows = 3, cols = 6, sentence = ["a", "bcd", "e"]

Output:
2

Explanation:
a-bcd-
e-a---
bcd-e-

The character '-' signifies an empty space on the screen.
```

Example 3:

Input:
rows = 4, cols = 5, sentence = ["I", "had", "apple", "pie"]

Output:
1

Explanation:
I-had
apple
pie-I
had--

The character '-' signifies an empty space on the screen.

The answers will be available soon! Meanwhile you can go check out the answers in the discussion forum so far.

From Leetcoder.

Strong Password Checker

A password is considered strong if below conditions are all met:

1. It has at least 6 characters and at most 20 characters.
2. It must contain at least one lowercase letter, at least one uppercase letter, and at least one digit.
3. It must NOT contain three repeating characters in a row ("...aaa..." is weak, but "...aa...a..." is strong, assuming other conditions are met).

Write a function strongPasswordChecker(s), that takes a string s as input, and return the **MINIMUM** change required to make s a strong password. If s is already strong, return 0.

Insertion, deletion or replace of any one character are all considered as one change.

Solution 1

I've separated the problem into three cases:
(1) `s.length() < 6`
(2) `6 <= s.length() <= 20`
(3) `s.length() > 20`

Let's look at case (1) first. If `s.length() < 6`, we know we have room to insert some more letters into `s`. Question is how to use the insertions effectively to reduce the number of potential replacements. I'm using a greedy approach for this one: I'm inserting one char between the second and third chars whenever I see a repetition of 3 letters as substring.

e.g. Say we have room to insert some chars in string and we see a substring of `"aaaa"`. I'll insert a `'B'` to make it `"aaBaa"` to break the 3-char repetition, thus reducing potential replacement by 1. And we'll do this until we can't insert any more chars into `s`. When we reach this point, we'll start dealing with case (2)

For case (2), I still follow a greedy approach. I'm simply searching for 3-char repetitions, and replacing one of the chars to break the repetition.
e.g. If we see a substring of `"aaaa"`, we'll make it `"aaBa"`.

My code deals with (1) and (2) together as `s.length() <= 20`.

Case (3) is a little bit tricky because simple greedy doesn't work any more.
When `s.length() > 20`, we want to delete some chars instead of inserting chars to reduce potential replacements. Question is the same: how to do it effectively? Let's do some observations here:

Say `len` is the length of each repetition.
(a) `len % 3` only has three possible values, namely 0, 1 and 2.
(b) Minimum number of replacements needed to break each repetition is `len / 3`.
(c) Based on (a) and (b), we know that deletion can reduce replacements only if the deletion can change the value of `len / 3`
(d) Based on (c), we know if we want to reduce 1 replacement, we need 1 deletion for `len % 3 == 0`, and 2 deletions for `len % 3 == 1`, and 3 deletions for `len % 3 == 2`.

Given above observations, I simply implemented the solution to do (d).

Also note that missing of upper case char, lower case char, or digit can always be resolved by insertion or replacement.

I've pasted two versions of the solutions below, with and without comments, for easier reference.

Without comments:

```
class Solution {
public:
    int strongPasswordChecker(string s) {
        int deleteTarget = max(0, (int)s.length() - 20), addTarget = max(0, 6 - (
int)s.length());
        int toDelete = 0, toAdd = 0, toReplace = 0, needUpper = 1, needLower = 1,
needDigit = 1;

        for (int l = 0, r = 0; r < s.length(); r++) {
            if (isupper(s[r])) { needUpper = 0; }
            if (islower(s[r])) { needLower = 0; }
            if (isdigit(s[r])) { needDigit = 0; }

            if (r - l == 2) {
                if (s[l] == s[l + 1] && s[l + 1] == s[r]) {
                    if (toAdd < addTarget) { toAdd++, l = r; }
                    else { toReplace++, l = r + 1; }
                } else { l++; }
            }
        }
        if (s.length() <= 20) { return max(addTarget + toReplace, needUpper + nee
dLower + needDigit); }

        toReplace = 0;
        vector<unordered_map<int, int>> lenCnts(3);
        for (int l = 0, r = 0, len; r <= s.length(); r++) {
            if (r == s.length() || s[l] != s[r]) {
                if ((len = r - l) > 2) { lenCnts[len % 3][len]++; }
                l = r;
            }
        }

        for (int i = 0, numLetters, dec; i < 3; i++) {
            for (auto it = lenCnts[i].begin(); it != lenCnts[i].end(); it++) {
                if (i < 2) {
                    numLetters = i + 1, dec = min(it->second, (deleteTarget - toD
elete) / numLetters);
                    toDelete += dec * numLetters, it->second -= dec;

                    if (it->first - numLetters > 2) { lenCnts[2][it->first - numL
etters] += dec; }
                }
                toReplace += (it->second) * ((it->first) / 3);
            }
        }

        int dec = (deleteTarget - toDelete) / 3;
        toReplace -= dec, toDelete -= dec * 3;
        return deleteTarget + max(toReplace, needUpper + needLower + needDigit);
    }
};
```

With comments:

```
class Solution {
public:
    int strongPasswordChecker(string s) {
        int deleteTarget = max(0, (int)s.length() - 20), addTarget = max(0, 6 - (
int)s.length());
        int toDelete = 0, toAdd = 0, toReplace = 0, needUpper = 1, needLower = 1,
needDigit = 1;

        ///////////////////////////////////
        // For cases of s.length() <= 20 //
        ///////////////////////////////////
        for (int l = 0, r = 0; r < s.length(); r++) {
            if (isupper(s[r])) { needUpper = 0; }
            if (islower(s[r])) { needLower = 0; }
            if (isdigit(s[r])) { needDigit = 0; }

            if (r - l == 2) {                                   // if it's a thr
ee-letter window
                if (s[l] == s[l + 1] && s[l + 1] == s[r]) {     // found a three-
repeating substr
                    if (toAdd < addTarget) { toAdd++, l = r; }  // insert letter
to break repetition if possible
                    else { toReplace++, l = r + 1; }            // replace curre
nt word to avoid three repeating chars
                } else { l++; }                                 // keep the wind
ow with no more than 3 letters
            }
        }
        if (s.length() <= 20) { return max(addTarget + toReplace, needUpper + nee
dLower + needDigit); }

        ////////////////////////////////////
        // For cases of s.length() > 20 //
        ////////////////////////////////////
        toReplace = 0;                                          // reset toRepla
ce
        vector<unordered_map<int, int>> lenCnts(3);             // to record repe
titions with (length % 3) == 0, 1 or 2
        for (int l = 0, r = 0, len; r <= s.length(); r++) {     // record all rep
etion frequencies
            if (r == s.length() || s[l] != s[r]) {
                if ((len = r - l) > 2) { lenCnts[len % 3][len]++; } // we only ca
re about repetions with length >= 3
                l = r;
            }
        }

        /*
            Use deletions to minimize replacements, following below orders:
            (1) Try to delete one letter from repetitions with (length % 3) == 0.
Each deletion decreases replacement by 1
            (2) Try to delete two letters from repetitions with (length % 3) == 1
. Each deletion decreases repalcement by 1
            (3) Try to delete multiple of three letters from repetions with (leng
th % 3) == 2. Each deletion (of three
            letters) decreases repalcements by 1
```

```
        */
        for (int i = 0, numLetters, dec; i < 3; i++) {
            for (auto it = lenCnts[i].begin(); it != lenCnts[i].end(); it++) {
                if (i < 2) {
                    numLetters = i + 1, dec = min(it->second, (deleteTarget - toD
elete) / numLetters);
                    toDelete += dec * numLetters;               // dec is the num
ber of repetitions we'll delete from
                    it->second -= dec;                          // update number
of repetitions left

                    // after letters deleted, it fits in the group where (length
% 3) == 2
                    if (it->first - numLetters > 2) { lenCnts[2][it->first - numL
etters] += dec; }
                }

                // record number of replacements needed
                // note if len is the length of repetition, we need (len / 3) num
ber of replacements
                toReplace += (it->second) * ((it->first) / 3);
            }
        }

        int dec = (deleteTarget - toDelete) / 3;                // try to delete
multiple of three letters as many as possible
        toReplace -= dec, toDelete -= dec * 3;
        return deleteTarget + max(toReplace, needUpper + needLower + needDigit);
    }
};
```

written by soamaaazing original link here

Solution 2

There are 2 wrong test cases which I list at the beginning.

The general idea is to record some states, and calculate the edit distance at the end. All detail are explained in the comments.

```java
public class Solution {
    public int strongPasswordChecker(String s) {

        if(s.length()<2) return 6-s.length();

        //Initialize the states, including current ending character(end), existen
ce of lowercase letter(lower), uppercase letter(upper), digit(digit) and number o
f replicates for ending character(end_rep)
        char end = s.charAt(0);
        boolean upper = end>='A'&&end<='Z', lower = end>='a'&&end<='z', digit = e
nd>='0'&&end<='9';

        //Also initialize the number of modification for repeated characters, tot
al number needed for eliminate all consequnce 3 same character by replacement(cha
nge), and potential maximun operation of deleting characters(delete). Note delete
[0] means maximum number of reduce 1 replacement operation by 1 deletion operatio
n, delete[1] means maximun number of reduce 1 replacement by 2 deletion operation
, delete[2] is no use here.
        int end_rep = 1, change = 0;
        int[] delete = new int[3];

        for(int i = 1;i<s.length();++i){
            if(s.charAt(i)==end) ++end_rep;
            else{
                change+=end_rep/3;
                if(end_rep/3>0) ++delete[end_rep%3];
                //updating the states
                end = s.charAt(i);
                upper = upper||end>='A'&&end<='Z';
                lower = lower||end>='a'&&end<='z';
                digit = digit||end>='0'&&end<='9';
                end_rep = 1;
            }
        }
        change+=end_rep/3;
        if(end_rep/3>0) ++delete[end_rep%3];

        //The number of replcement needed for missing of specific character(lower
/upper/digit)
        int check_req = (upper?0:1)+(lower?0:1)+(digit?0:1);

        if(s.length()>20){
            int del = s.length()-20;

            //Reduce the number of replacement operation by deletion
            if(del<=delete[0]) change-=del;
            else if(del-delete[0]<=2*delete[1]) change-=delete[0]+(del-delete[0])
/2;
            else change-=delete[0]+delete[1]+(del-delete[0]-2*delete[1])/3;

            return del+Math.max(check_req,change);
        }
        else return Math.max(6-s.length(), Math.max(check_req, change));
    }
}
```

The author is already fixed all the test cases.

written by danzhutest original link here

Solution 3

The basic principle is straightforward: if we want to make MINIMUM changes to turn `s` into a strong password, each change made should fix as many problems as possible.

So to start, let's first identify all the problems in the input string `s` and list what changes are suitable for righting each of them. To clarify, each change should be characterized by at least two parts: the **type of operation** it takes and the **position in the string** where the operation is applied (`Note`: Ideally we should also include the characters involved in the operation and the "power" of each operation for eliminating problems but they turn out to be partially relevant so I will mention them only when appropriate).

1. `Length problem`: if the total length is **less than 6**, the change that should be made is (**insert, any position**), which reads as "the operation is insertion and it can be applied to anywhere in the string". If the total length is **greater than 20**, then the change should be (**delete, any position**).
2. `Missing letter or digit`: if any of the lowercase/uppercase letters or digits is missing, we can do either (**insert, any position**) or (**replace, any position**) to correct it. (`Note` here the characters for insertion or replacement can only be those missing.)
3. `Repeating characters`: for repeating characters, all three operations are allowed but the positions where they can be applied are limited within the repeating characters. For example, to fix `"aaaaa"`, we can do one replacement (replace the middle `'a'`) or two insertions (one after the second `'a'` and one after the fourth `'a'`) or three deletions (delete any of the three `'a'`s). So the possible changes are (**replace, repeating characters**), (**insert, repeating characters**), (**delete, repeating characters**). (`Note` however the "power" of each operation for fixing the problem are different -- replacement is the strongest while deletion is the weakest.)

All right, what's next? If we want a change to eliminate as many problems as it can, it must be shared among the possible solutions to each problem it can fix. So our task is to find out possible overlapping among the changes for fixing each problem.

Since there are most (three) changes allowed for the third problem, we may start from combinations `first problem & third problem` and `second problem & third problem`. It's not too hard to conclude that any change that can fix the first or second problems is also able to fix the third one (since the type of operation here is irrelevant, we are free to choose the position of the operation to match those of the repeating characters). For combination `first problem & second problem`, depending on the length of the string, there will be overlapping if length is less than 6 or no overlapping if length is greater than 20.

From the analyses above, it seems worthwhile to distinguish between the two cases: when the input string is too short or too long.

For the former case, it can be shown that the total changes needed to fix the first and second problems always outnumber those for the third one. Since whatever change used fixing the first two problems can also correct the third one, we may concern ourselves with only the first two. Also as there is overlapping between the changes for fixing the first two problems, we will prefer those overlapping ones, i.e. (**insert, any position**). Another point is that the characters involved in the operation matters now. To fix the first problem, only those missing characters can be inserted while for the second condition, it can be any character. Therefore correcting the first problem takes precedence over the second one.

For the latter case, there is overlapping between the**first & third** and **second & third** problems, so those overlapping changes will be taken, i.e., first problem => (**delete, any position**), second problem => (**replace, any position**). The reason not to use (**insert, any position**) for the second problem is that it contradicts the changes made to the first problem (therefore has the tendency to cancel its effects). After fixing the first two problems, what operation(s) should we choose for the third one?

Now the "power" of each operation for eliminating problems comes into play. For the third problem, the "power" of each operation will be measured by the maximum number of repeating characters it is able to get rid of. For example, one replacement can eliminate at most `5` repeating characters while insertion and deletion can do at most `4` and `3`, respectively. In this case, we say replacement has more "power" than insertion or deletion. Intuitively the more "powerful" the operation is, the less number of changes is needed for correcting the problem. Therefore (**replace, repeating characters**) triumphs in terms of fixing the third problem.

Furthermore, another very interesting point shows up when the "power" of operation is taken into consideration (And thank **yicui** for pointing it out). As I mentioned that there is overlapping between changes made for fixing the first two problems and for the third one, which means the operations chosen above for the first two problems will also be applied to the third one. For the second problem with change chosen as (**replace, any position**), we have no problem adapting it so that it coincides with the optimal change (**replace, repeating characters**) made for the third problem. However, there is no way to do the same for the first problem with change (**delete, any position**). We have a conflict now!

How do we reconcile it? The trick is that for a sequence of repeating characters of length `k` (`k >= 3`), instead of turning it all the way into a sequence of length `2` (so as to fix the repeating character problem) by the change (**delete, any position**), we will first reduce its length to (`3m + 2`), where (`3m + 2`) is the largest integer of the form yet no more than `k`. That is to say, if `k` is a multiple of `3`, we apply once such change so its length will become (`k - 1`); else if `k` is a multiple of `3` plus `1`, we apply twice such change to cut its length down to (`k - 2`), provided we have more such changes to spare (be careful here as we need at least two changes but the remaining available changes may be less than that, so we should stick to the smaller one: `2` or the remaining available changes). The reason is that the optimal change (**replace, repeating characters**) for the third problem will be most "powerful"

when the total length of the repeating characters is of this form. Of course, if we still have more changes (**delete, any position**) to do after that, then we are free to turn the repeating sequence all the way into a sequence of length `2`.

Here is the java program based on the above analyses. Both time and space complexity is `O(n)`. Not sure if we can reduce the space down to `O(1)` by computing the `arr` array on the fly. A quick explanation is given at the end.

```java
public int strongPasswordChecker(String s) {
    int res = 0, a = 1, A = 1, d = 1;
    char[] carr = s.toCharArray();
    int[] arr = new int[carr.length];

    for (int i = 0; i < arr.length;) {
        if (Character.isLowerCase(carr[i])) a = 0;
        if (Character.isUpperCase(carr[i])) A = 0;
        if (Character.isDigit(carr[i])) d = 0;

        int j = i;
        while (i < carr.length && carr[i] == carr[j]) i++;
        arr[j] = i - j;
    }

    int total_missing = (a + A + d);

    if (arr.length < 6) {
        res += total_missing + Math.max(0, 6 - (arr.length + total_missing));

    } else {
        int over_len = Math.max(arr.length - 20, 0), left_over = 0;
        res += over_len;

        for (int k = 1; k < 3; k++) {
            for (int i = 0; i < arr.length && over_len > 0; i++) {
                if (arr[i] < 3 || arr[i] % 3 != (k - 1)) continue;
                arr[i] -= Math.min(over_len, k);
                over_len -= k;
            }
        }

        for (int i = 0; i < arr.length; i++) {
            if (arr[i] >= 3 && over_len > 0) {
                int need = arr[i] - 2;
                arr[i] -= over_len;
                over_len -= need;
            }

            if (arr[i] >= 3) left_over += arr[i] / 3;
        }

        res += Math.max(total_missing, left_over);
    }

    return res;
}
```

A quick explanation of the program:

1. `res` denotes the minimum changes; `a`, `A` and `d` are the number of missing lowercase letters, uppercase letters and digits, respectively; `arr` is an integer array whose element will be the number of repeating characters starting at the corresponding position in the string.

2. In the following loop we fill in the values for `a`, `A`, `d` and `arr` to identify the problems for each condition. The total number of missing characters `total_missing` will be the summation of `a`, `A`, `d` and fixing this problem takes at least `total_missing` changes.
3. We then distinguish the two cases when the string is too short or too long. If it is too short, we pad its length to at least `6` (note in this case we've already inserted `total_missing` characters so the new length is the summation of the original length and `total_missing`).
4. Otherwise, to fix the first condition, we need to delete `over_len` (number of surplus characters) characters. Since fixing the first problem also corrects the third one, we need to get rid of those parts from the `arr` array. And as I mentioned, we need to first turn all numbers in the `arr` array greater than `2` into the form of (`3m + 2`) and then reduce them all the way to `2` if `over_len` is still greater than `0`. After that, we need to replace `total_missing` characters to fix the second problem, which also fixes part (or all) of the third problem. Therefore we only need to take the larger number of changes needed for fixing the second problem (which is `total_missing`) and for the third one (which is `left_over`, since it is the number of changes still needed after fixing the first problem).

written by fun4LeetCode original link here

From Leetcoder.

Maximum XOR of Two Numbers in an Array

Given a **non-empty** array of numbers, $a_0, a_1, a_2, \ldots, a_{n-1}$, where $0 \le a_i$ 31.

Find the maximum result of a_i XOR a_j, where $0 \le i, j < n$.

Could you do this in $O(n)$ runtime?

Example:

```
Input: [3, 10, 5, 25, 2, 8]

Output: 28

Explanation: The maximum result is 5 ^ 25 = 28.
```

Solution 1

```
public class Solution {
    public int findMaximumXOR(int[] nums) {
        int max = 0, mask = 0;
        for(int i = 31; i >= 0; i--){
            mask = mask | (1 << i);
            Set<Integer> set = new HashSet<>();
            for(int num : nums){
                set.add(num & mask);
            }
            int tmp = max | (1 << i);
            for(int prefix : set){
                if(set.contains(tmp ^ prefix)) {
                    max = tmp;
                    break;
                }
            }
        }
        return max;
    }
}
```

written by tangx668 original link here

Solution 2

```
def findMaximumXOR(self, nums):
    answer = 0
    for i in range(32)[::-1]:
        answer <<= 1
        prefixes = {num >> i for num in nums}
        answer += any(answer^1 ^ p in prefixes for p in prefixes)
    return answer
```

Build the answer bit by bit from left to right. Let's say we already know the largest first seven bits we can create. How to find the largest first eight bits we can create? Well it's that maximal seven-bits prefix followed by 0 or 1. Append 0 and then try to create the 1 one (i.e., `answer ^ 1`) from two eight-bits prefixes from `nums`. If we can, then change that 0 to 1.

written by StefanPochmann original link here

Solution 3

```
class Trie {
    Trie[] children;
    public Trie() {
        children = new Trie[2];
    }
}

public int findMaximumXOR(int[] nums) {
    if(nums == null || nums.length == 0) {
        return 0;
    }
    // Init Trie.
    Trie root = new Trie();
    for(int num: nums) {
        Trie curNode = root;
        for(int i = 31; i >= 0; i --) {
            int curBit = (num >>> i) & 1;
            if(curNode.children[curBit] == null) {
                curNode.children[curBit] = new Trie();
            }
            curNode = curNode.children[curBit];
        }
    }
    int max = Integer.MIN_VALUE;
    for(int num: nums) {
        Trie curNode = root;
        int curSum = 0;
        for(int i = 31; i >= 0; i --) {
            int curBit = (num >>> i) & 1;
            if(curNode.children[curBit ^ 1] != null) {
                curSum += (1 << i);
                curNode = curNode.children[curBit ^ 1];
            }else {
                curNode = curNode.children[curBit];
            }
        }
        max = Math.max(curSum, max);
    }
    return max;
}
```

written by mywen1234 original link here

From Leetcoder.

Valid Word Square

Given a sequence of words, check whether it forms a valid word square.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where $0 \le k < \max(\text{numRows}, \text{numColumns})$.

Note:

1. The number of words given is at least 1 and does not exceed 500.
2. Word length will be at least 1 and does not exceed 500.
3. Each word contains only lowercase English alphabet `a-z`.

Example 1:

```
Input:
[
  "abcd",
  "bnrt",
  "crmy",
  "dtye"
]

Output:
true

Explanation:
The first row and first column both read "abcd".
The second row and second column both read "bnrt".
The third row and third column both read "crmy".
The fourth row and fourth column both read "dtye".

Therefore, it is a valid word square.
```

Example 2:

```
Input:
[
  "abcd",
  "bnrt",
  "crm",
  "dt"
]

Output:
true

Explanation:
The first row and first column both read "abcd".
The second row and second column both read "bnrt".
The third row and third column both read "crm".
The fourth row and fourth column both read "dt".

Therefore, it is a valid word square.
```

Example 3:

Input:
```
[
  "ball",
  "area",
  "read",
  "lady"
]
```

Output:
false

Explanation:
The third row reads "read" while the third column reads "lead".

Therefore, it is **NOT** a valid word square.

Solution 1

```
    def validWordSquare(self, words):
    """
    :type words: List[str]
    :rtype: bool
    """
    lenw = len(words)
    try:
        for i in range(lenw):
            lenj = len(words[i])
            for j in range(lenj):
                if words[i][j] != words[j][i]:
                    return False
        return True
    except IndexError:
        return False
```

written by leakey original link here

Solution 2

Solution:

```
def validWordSquare(self, words):
    return map(None, *words) == map(None, *map(None, *words))
```

Or saving some work but taking two lines:

```
def validWordSquare(self, words):
    t = map(None, *words)
    return t == map(None, *t)
```

Explanation:

The `map(None, ...)` transposes the "matrix", filling missing spots with `None`. For example:

```
["abc",            [('a', 'd', 'f'),
 "de",     =>       ('b', 'e', None),
 "f"]               ('c', None, None)]
```

And then I just need to check whether transposing it once more changes it.

written by StefanPochmann original link here

Solution 3

```
   bool validWordSquare(vector<string>& words) {
        for(int i = 0; i < words.size(); ++i) {
            for(int j = 0; j < words[i].size(); ++j)             {
                if(j >= words.size() || words[j].size() <= i || words[j][i] != wo
rds[i][j])
                    return false;
            }
        }
        return true;
   }
```

written by i9r0k original link here

From Leetcoder.

Reconstruct Original Digits from English

Given a **non-empty** string containing an out-of-order English representation of digits `0-9`, output the digits in ascending order.

Note:

1. Input contains only lowercase English letters.
2. Input is guaranteed to be valid and can be transformed to its original digits. That means invalid inputs such as "abc" or "zerone" are not permitted.
3. Input length is less than 50,000.

Example 1:

```
Input: "owoztneoer"

Output: "012"
```

Example 2:

```
Input: "fviefuro"

Output: "45"
```

Solution 1

The idea is:

for zero, it's the only word has letter 'z',
for two, it's the only word has letter 'w',
.....
so we only need to count the unique letter of each word, Coz the input is always valid.

Code:

```
public String originalDigits(String s) {
    int[] count = new int[10];
    for (int i = 0; i < s.length(); i++){
        char c = s.charAt(i);
        if (c == 'z') count[0]++;
        if (c == 'w') count[2]++;
        if (c == 'x') count[6]++;
        if (c == 's') count[7]++; //7-6
        if (c == 'g') count[8]++;
        if (c == 'u') count[4]++;
        if (c == 'f') count[5]++; //5-4
        if (c == 'h') count[3]++; //3-8
        if (c == 'i') count[9]++; //9-8-5-6
        if (c == 'o') count[1]++; //1-0-2-4
    }
    count[7] -= count[6];
    count[5] -= count[4];
    count[3] -= count[8];
    count[9] = count[9] - count[8] - count[5] - count[6];
    count[1] = count[1] - count[0] - count[2] - count[4];
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i <= 9; i++){
        for (int j = 0; j < count[i]; j++){
            sb.append(i);
        }
    }
    return sb.toString();
}
```

written by markieff original link here

Solution 2

```
public class Solution {
    public String originalDigits(String s) {
        if(s==null || s.length()==0) return "";
        int[] count = new int[128];
        for(int i=0;i<s.length();i++)  count[s.charAt(i)]++;
        int[] num = new int[10];
        num[0] = count['z'];
        num[2] = count['w'];
        num[6] = count['x'];
        num[8] = count['g'];
        num[7] = count['s']-count['x'];
        num[5] = count['v']-count['s']+count['x'];
        num[4] = count['u'];
        num[3] = count['h']-count['g'];
        num[1] = count['o']-count['z']-count['w']-count['u'];
        num[9] = count['i']-count['x']-count['g']-count['v']+count['s']-count['x'
];
        String ret = new String();
        for(int i=0;i<10;i++)
            for(int j=num[i];j>0;j--) ret += String.valueOf(i);
        return ret;
    }
}
```

written by YuTingLiu original link here

Solution 3

The **even** digits all have a unique letter while the**odd** digits all don't:

`zero` : Only digit with `z`
`two` : Only digit with `w`
`four` : Only digit with `u`
`six` : Only digit with `x`
`eight` : Only digit with `g`

The odd ones for easy looking, each one's letters all also appear in other digit words: `one`, `three`, `five`, `seven`, `nine`

written by StefanPochmann original link here

From Leetcoder.

Longest Repeating Character Replacement

Given a string that consists of only uppercase English letters, you can replace any letter in the string with another letter at most *k* times. Find the length of a longest substring containing all repeating letters you can get after performing the above operations.

Note:
Both the string's length and *k* will not exceed 10^4.

Example 1:

```
Input:
s = "ABAB", k = 2

Output:
4

Explanation:
Replace the two 'A's with two 'B's or vice versa.
```

Example 2:

```
Input:
s = "AABABBA", k = 1

Output:
4

Explanation:
Replace the one 'A' in the middle with 'B' and form "AABBBBA".
The substring "BBBB" has the longest repeating letters, which is 4.
```

Solution 1

The problem says that we can make at most k changes to the string (any character can be replaced with any other character). So, let's say there were no constraints like the k. Given a string convert it to a string with all same characters with minimal changes. The answer to this is

> length of the entire string - number of times of the maximum occurring character in the string

Given this, we can apply the at most k changes constraint and maintain a sliding window such that

> (length of substring - number of times of the maximum occurring character in the substring) <= k

```
class Solution {
public:
    int characterReplacement(string s, int k) {
        vector<int> counts(26, 0);
        int start = 0;
        int maxCharCount = 0;
        int n = s.length();
        int result = 0;
        for(int end = 0; end < n; end++){
            counts[s[end]-'A']++;
            if(maxCharCount < counts[s[end]-'A']){
                maxCharCount = counts[s[end]-'A'];
            }
            while(end-start-maxCharCount+1 > k){
                counts[s[start]-'A']--;
                start++;
                for(int i = 0; i < 26; i++){
                    if(maxCharCount < counts[i]){
                        maxCharCount = counts[i];
                    }
                }
            }
            result = max(result, end-start+1);
        }
        return result;
    }
};
```

written by harunrashidanver original link here

Solution 2

```
    public int characterReplacement(String s, int k) {
        int len = s.length();
        int[] count = new int[26];
        int start = 0, maxCount = 0, maxLength = 0;
        for (int end = 0; end < len; end++) {
            maxCount = Math.max(maxCount, ++count[s.charAt(end) - 'A']);
            while (end - start + 1 - maxCount > k) {
                count[s.charAt(start) - 'A']--;
                start++;
            }
            maxLength = Math.max(maxLength, end - start + 1);
        }
        return maxLength;
    }
```

There's no edge case for this question. The initial step is to extend the window to its limit, that is, the longest we can get to with maximum number of modifications. Until then the variable **start** will remain at 0.

Then as **end** increase, the whole substring from 0 to**end** will violate the rule, so we need to update **start** accordingly (slide the window). We move**start** to the right until the whole string satisfy the constraint again. Then each time we reach such situation, we update our max length.

written by Joshua924 original link here

Solution 3

Based on the Python solution by @dalwise. Use a sliding window `s[i:j]`, always add the new character, and remove the first window character if the extension isn't ok. So in each step, either extend the window by one or move it by one.

```
int characterReplacement(string s, int k) {
    int i = 0, j = 0, ctr[91] = {};
    while (j < s.size()) {
        ctr[s[j++]]++;
        if (j-i - *max_element(ctr+65, ctr+91) > k)
            ctr[s[i++]]--;
    }
    return j - i;
}
```

written by StefanPochmann original link here

From Leetcoder.

Word Squares

Given a set of words **(without duplicates)**, find all word squares you can build from them.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where $0 \le k < \max(\text{numRows}, \text{numColumns})$.

For example, the word sequence `["ball","area","lead","lady"]` forms a word square because each word reads the same both horizontally and vertically.

```
b a l l
a r e a
l e a d
l a d y
```

Note:

1. There are at least 1 and at most 1000 words.
2. All words will have the exact same length.
3. Word length is at least 1 and at most 5.
4. Each word contains only lowercase English alphabet `a-z`.

Example 1:

```
Input:
["area","lead","wall","lady","ball"]

Output:
[
  [ "wall",
    "area",
    "lead",
    "lady"
  ],
  [ "ball",
    "area",
    "lead",
    "lady"
  ]
]

Explanation:
The output consists of two word squares. The order of output does not matter (just
the order of words in each word square matters).
```

Example 2:

Input:
```
["abat","baba","atan","atal"]
```

Output:
```
[
  [ "baba",
    "abat",
    "baba",
    "atan"
  ],
  [ "baba",
    "abat",
    "baba",
    "atal"
  ]
]
```

Explanation:
The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

Solution 1

My first approach is brute-force, try every possible word sequences, and use the solution of Problem 422 (https://leetcode.com/problems/valid-word-square/) to check each sequence. This solution is straightforward, but too slow (TLE).

A better approach is to check the validity of the word square while we build it.
Example: `["area","lead","wall","lady","ball"]`
We know that the sequence contains 4 words because the length of each word is 4.
Every word can be the first word of the sequence, let's take `"wall"` for example.
Which word could be the second word? Must be a word start with `"a"` (therefore `"area"`), because it has to match the second letter of word `"wall"`.
Which word could be the third word? Must be a word start with `"le"` (therefore `"lead"`), because it has to match the third letter of word `"wall"` and the third letter of word `"area"`.
What about the last word? Must be a word start with `"lad"` (therefore `"lady"`).
For the same reason above.

The picture below shows how the prefix are matched while building the sequence.

In order for this to work, we need to fast retrieve all the words with a given **prefix**.
There could be 2 ways doing this:

1. Using a hashtable, key is **prefix**, value is a list of words with that prefix.
2. Trie, we store a list of words with the **prefix** on each trie node.

The implemented below uses Trie.

```
public class Solution {
    class TrieNode {
        List<String> startWith;
        TrieNode[] children;

        TrieNode() {
            startWith = new ArrayList<>();
            children = new TrieNode[26];
        }
    }

    class Trie {
        TrieNode root;

        Trie(String[] words) {
            root = new TrieNode();
            for (String w : words) {
                TrieNode cur = root;
                for (char ch : w.toCharArray()) {
                    int idx = ch - 'a';
                    if (cur.children[idx] == null)
                        cur.children[idx] = new TrieNode();
                    cur.children[idx].startWith.add(w);
                    cur = cur.children[idx];
```

```
                }
            }
        }

        List<String> findByPrefix(String prefix) {
            List<String> ans = new ArrayList<>();
            TrieNode cur = root;
            for (char ch : prefix.toCharArray()) {
                int idx = ch - 'a';
                if (cur.children[idx] == null)
                    return ans;

                cur = cur.children[idx];
            }
            ans.addAll(cur.startWith);
            return ans;
        }
    }

    public List<List<String>> wordSquares(String[] words) {
        List<List<String>> ans = new ArrayList<>();
        if (words == null || words.length == 0)
            return ans;
        int len = words[0].length();
        Trie trie = new Trie(words);
        List<String> ansBuilder = new ArrayList<>();
        for (String w : words) {
            ansBuilder.add(w);
            search(len, trie, ans, ansBuilder);
            ansBuilder.remove(ansBuilder.size() - 1);
        }

        return ans;
    }

    private void search(int len, Trie tr, List<List<String>> ans,
            List<String> ansBuilder) {
        if (ansBuilder.size() == len) {
            ans.add(new ArrayList<>(ansBuilder));
            return;
        }

        int idx = ansBuilder.size();
        StringBuilder prefixBuilder = new StringBuilder();
        for (String s : ansBuilder)
            prefixBuilder.append(s.charAt(idx));
        List<String> startWith = tr.findByPrefix(prefixBuilder.toString());
        for (String sw : startWith) {
            ansBuilder.add(sw);
            search(len, tr, ans, ansBuilder);
            ansBuilder.remove(ansBuilder.size() - 1);
        }
    }
}
```

written by lzb700m original link here

Solution 2

Python Solution (accepted in ~870 ms)

```
def wordSquares(self, words):
    n = len(words[0])
    fulls = collections.defaultdict(list)
    for word in words:
        for i in range(n):
            fulls[word[:i]].append(word)
    def build(square):
        if len(square) == n:
            squares.append(square)
            return
        for word in fulls[''.join(zip(*square)[len(square)])]:
            build(square + [word])
    squares = []
    for word in words:
        build([word])
    return squares
```

Explanation

I try every word for the first row. For each of them, try every fitting word for the second row. And so on. The first few rows determine the first few columns and thus determine how the next row's word must start. For example:

```
wall      Try words      wall                      wall                      wall
a...   => starting  =>   area      Try words       area                      area
l...      with "a"       le..   => starting  =>    lead      Try words       lead
l...                     la..      with "le"       lad.   => starting   =>   lady
                                                             with "lad"
```

For quick lookup, my `fulls` dictionary maps prefixes to lists of words who have that prefix.

C++ Solution (accepted in ~180 ms)

```
class Solution {
public:
    vector<vector<string>> wordSquares(vector<string>& words) {
        n = words[0].size();
        square.resize(n);
        for (string word : words)
            for (int i=0; i<n; i++)
                fulls[word.substr(0, i)].push_back(word);
        build(0);
        return squares;

    }
    int n;
    unordered_map<string, vector<string>> fulls;
    vector<string> square;
    vector<vector<string>> squares;
    void build(int i) {
        if (i == n) {
            squares.push_back(square);
            return;
        }
        string prefix;
        for (int k=0; k<i; k++)
            prefix += square[k][i];
        for (string word : fulls[prefix]) {
            square[i] = word;
            build(i + 1);
        }
    }
};
```

written by StefanPochmann original link here

Solution 3

Optimized to 70ms by pre-size the vec instead of push_back and pop_back

```cpp
class Solution {
public:
    struct TrieNode {
        vector<int> indexs;
        vector<TrieNode*> children;
        TrieNode() {
            children.resize(26, nullptr);
        }
    };

    TrieNode* buildTrie(vector<string>& words) {
        TrieNode* root = new TrieNode();
        for (int j = 0; j < words.size(); j++) {
            TrieNode* t = root;
            for (int i = 0; i < words[j].size(); i++) {
                if (!t->children[words[j][i] - 'a'])
                    t->children[words[j][i] - 'a'] = new TrieNode();
                t = t->children[words[j][i] - 'a'];
                t->indexs.push_back(j);
            }
        }
        return root;
    }

    vector<vector<string>> res;
    vector<string> vec;
    void backtrack(vector<string>& words, int level, TrieNode* root) {
        if (level >= words[0].size()) {
            res.push_back(vec);
            return;
        }
        string str = "";
        for (int i = 0; i < level; i++)
            str += vec[i][level];
        TrieNode* t = root;
        for (int i = 0; i < str.size(); i++)
            if (!(t = t->children[str[i] - 'a'])) return;
        for (auto index : t->indexs) {
            vec[level] = words[index];
            backtrack(words, level + 1, root);
        }
    }

    vector<vector<string>> wordSquares(vector<string>& words) {
        if (words.empty()) return res;
        TrieNode* root = buildTrie(words);
        vec.resize((int)words[0].size());
        for (auto& word : words) {
            vec[0] = word;
            backtrack(words, 1, root);
        }
        return res;
    }
};
```

written by zyoppy008 original link here

From Leetcoder.

All O`one Data Structure

Implement a data structure supporting the following operations:

1. Inc(Key) - Inserts a new key with value 1. Or increments an existing key by 1. Key is guaranteed to be a **non-empty** string.
2. Dec(Key) - If Key's value is 1, remove it from the data structure. Otherwise decrements an existing key by 1. If the key does not exist, this function does nothing. Key is guaranteed to be a **non-empty** string.
3. GetMaxKey() - Returns one of the keys with maximal value. If no element exists, return an empty string `""`.
4. GetMinKey() - Returns one of the keys with minimal value. If no element exists, return an empty string `""`.

Challenge: Perform all these in O(1) time complexity.

Solution 1

The main idea is to maintain an ordered two-dimensional doubly-linked list (let's call it matrix for convenience), of which each row is corresponding to a value and all of the keys in the same row have the same value.

Suppose we get the following key-value pairs after some increment operations. ("A": 4 means "A" is increased four times so its value is 4, and so on.)

```
"A": 4, "B": 4, "C": 2, "D": 1
```

Then one possible matrix may look like this:

```
row0: val = 4, strs = {"A", "B"}
row1: val = 2, strs = {"C"}
row2: val = 1, strs = {"D"}
```

If we can guarantee the rows are in descending order in terms of value, then GetMaxKey()/GetMinKey() will be easy to implement in O(1) time complexity. Because the first key in the first row will always has the maximal value, and the first key in the last row will always has the minimal value.

Once a key is increased, we move the key from current row to last row if last_row.val = current_row.val + 1. Otherwise, we insert a new row before current row with vallue current_row.val + 1, and move the key to to the new row. The logic of decrement operation is similar. Obviously, by doing this, the rows will keep its descending order.

For example, after Inc("D"), the matrix will become

```
row0: val = 4, strs = {"A", "B"}
row1: val = 2, strs = {"C", "D"}
```

Inc("D") again

```
row0: val = 4, strs = {"A", "B"}
row1: val = 3, strs = {"D"}
row2: val = 2, strs = {"C"}
```

Now the key problem is how to maintain the matrix in O(1) runtime when increase/decrease a key by 1.

The answer is hash map. By using a hash map to track the position of a key in the matrix, we can access a key in the matrix in O(1). And since we use linked list to store the matrix, thus insert/move operations will all be O(1).

The psudocode of Inc() is as follows(Dec() is similar).

```
if the key isn't in the matrix:
    if the matrix is empty or the value of the last row isn't 1:
        insert a new row with value 1 to the end of the matrix, and put the key i
n the new row;
    else:
        put the key in the last row of the matrix;
else:
    if the key is at the first row or last_row.value != current_row.value + 1:
        insert a new row before current row, with value current_row.value + 1, an
d move the key to the new row;
    else:
        move the key from current row to last row;
```

Here is the code.

```
class AllOne {
public:
    struct Row {
        list<string> strs;
        int val;
        Row(const string &s, int x) : strs({s}), val(x) {}
    };

    unordered_map<string, pair<list<Row>::iterator, list<string>::iterator>> strm
ap;
    list<Row> matrix;

    /** Initialize your data structure here. */
    AllOne() {

    }

    /** Inserts a new key <Key> with value 1. Or increments an existing key by 1.
*/
    void inc(string key) {
        if (strmap.find(key) == strmap.end()) {
            if (matrix.empty() || matrix.back().val != 1) {
                auto newrow = matrix.emplace(matrix.end(), key, 1);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
            else {
                auto newrow = --matrix.end();
                newrow->strs.push_front(key);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
        }
        else {
            auto row = strmap[key].first;
            auto col = strmap[key].second;
            auto lastrow = row;
            --lastrow;
            if (lastrow == matrix.end() || lastrow->val != row->val + 1) {
                auto newrow = matrix.emplace(row, key, row->val + 1);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
```

```
            else {
                auto newrow = lastrow;
                newrow->strs.push_front(key);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
            row->strs.erase(col);
            if (row->strs.empty()) matrix.erase(row);
        }
    }

    /** Decrements an existing key by 1. If Key's value is 1, remove it from the d
ata structure. */
    void dec(string key) {
        if (strmap.find(key) == strmap.end()) {
            return;
        }
        else {
            auto row = strmap[key].first;
            auto col = strmap[key].second;
            if (row->val == 1) {
                row->strs.erase(col);
                if (row->strs.empty()) matrix.erase(row);
                strmap.erase(key);
                return;
            }
            auto nextrow = row;
            ++nextrow;
            if (nextrow == matrix.end() || nextrow->val != row->val - 1) {
                auto newrow = matrix.emplace(nextrow, key, row->val - 1);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
            else {
                auto newrow = nextrow;
                newrow->strs.push_front(key);
                strmap[key] = make_pair(newrow, newrow->strs.begin());
            }
            row->strs.erase(col);
            if (row->strs.empty()) matrix.erase(row);
        }
    }

    /** Returns one of the keys with maximal value. */
    string getMaxKey() {
        return matrix.empty() ?  "" : matrix.front().strs.front();
    }

    /** Returns one of the keys with Minimal value. */
    string getMinKey() {
        return matrix.empty() ?  "" : matrix.back().strs.front();
    }
};
```

written by ivancjw original link here

Solution 2

For each value, I have a bucket with all keys which have that value. The buckets are in a list, sorted by value. That allows constant time insertion/erasure and iteration to the next higher/lower value bucket. A bucket stores its keys in a hash set for easy constant time insertion/erasure/check (see first two posts here if you're worried). I also have one hash map to look up which bucket a given key is in.

Based on a previously flawed Python attempt (I just couldn't find a good way to get an arbitrary element from a set) but also influenced by an earlier version of @Ren.W's solution. We ended up with quite similiar code, I guess there's not much room for creativity once you decide on the data types to hold the data.

```
class AllOne {
public:

    void inc(string key) {

        // If the key doesn't exist, insert it with value 0.
        if (!bucketOfKey.count(key))
            bucketOfKey[key] = buckets.insert(buckets.begin(), {0, {key}});

        // Insert the key in next bucket and update the lookup.
        auto next = bucketOfKey[key], bucket = next++;
        if (next == buckets.end() || next->value > bucket->value + 1)
            next = buckets.insert(next, {bucket->value + 1, {}});
        next->keys.insert(key);
        bucketOfKey[key] = next;

        // Remove the key from its old bucket.
        bucket->keys.erase(key);
        if (bucket->keys.empty())
            buckets.erase(bucket);
    }

    void dec(string key) {

        // If the key doesn't exist, just leave.
        if (!bucketOfKey.count(key))
            return;

        // Maybe insert the key in previous bucket and update the lookup.
        auto prev = bucketOfKey[key], bucket = prev--;
        bucketOfKey.erase(key);
        if (bucket->value > 1) {
            if (bucket == buckets.begin() || prev->value < bucket->value - 1)
                prev = buckets.insert(bucket, {bucket->value - 1, {}});
            prev->keys.insert(key);
            bucketOfKey[key] = prev;
        }

        // Remove the key from its old bucket.
        bucket->keys.erase(key);
        if (bucket->keys.empty())
            buckets.erase(bucket);
```

```
    }

    string getMaxKey() {
        return buckets.empty() ? "" : *(buckets.rbegin()->keys.begin());
    }

    string getMinKey() {
        return buckets.empty() ? "" : *(buckets.begin()->keys.begin());
    }

private:
    struct Bucket { int value; unordered_set<string> keys; };
    list<Bucket> buckets;
    unordered_map<string, list<Bucket>::iterator> bucketOfKey;
};
```

written by StefanPochmann original link here

Solution 3

Main idea is to maintain a list of Bucket's, each Bucket contains all keys with the same count.

1. `head` and `tail` can ensure both `getMaxKey()` and `getMaxKey()` be done in O(1).
2. `keyCountMap` maintains the count of keys, `countBucketMap` provides O(1) access to a specific Bucket with given count. Deleting and adding a Bucket in the Bucket list cost O(1), so both `inc()` and `dec()` take strict O(1) time.

```
public class AllOne {
    // maintain a doubly linked list of Buckets
    private Bucket head;
    private Bucket tail;
    // for accessing a specific Bucket among the Bucket list in O(1) time
    private Map<Integer, Bucket> countBucketMap;
    // keep track of count of keys
    private Map<String, Integer> keyCountMap;

    // each Bucket contains all the keys with the same count
    private class Bucket {
        int count;
        Set<String> keySet;
        Bucket next;
        Bucket pre;
        public Bucket(int cnt) {
            count = cnt;
            keySet = new HashSet<>();
        }
    }

    /** Initialize your data structure here. */
    public AllOne() {
        head = new Bucket(Integer.MIN_VALUE);
        tail = new Bucket(Integer.MAX_VALUE);
        head.next = tail;
        tail.pre = head;
        countBucketMap = new HashMap<>();
        keyCountMap = new HashMap<>();
    }

    /** Inserts a new key <Key> with value 1. Or increments an existing key by 1.
*/
    public void inc(String key) {
        if (keyCountMap.containsKey(key)) {
            changeKey(key, 1);
        } else {
            keyCountMap.put(key, 1);
            if (head.next.count != 1)
                addBucketAfter(new Bucket(1), head);
            head.next.keySet.add(key);
            countBucketMap.put(1, head.next);
        }
    }
```

```java
    /** Decrements an existing key by 1. If Key's value is 1, remove it from the d
ata structure. */
    public void dec(String key) {
        if (keyCountMap.containsKey(key)) {
            int count = keyCountMap.get(key);
            if (count == 1) {
                keyCountMap.remove(key);
                removeKeyFromBucket(countBucketMap.get(count), key);
            } else {
                changeKey(key, -1);
            }
        }
    }

    /** Returns one of the keys with maximal value. */
    public String getMaxKey() {
        return tail.pre == head ? "" : (String) tail.pre.keySet.iterator().next()
;
    }

    /** Returns one of the keys with Minimal value. */
    public String getMinKey() {
        return head.next == tail ? "" : (String) head.next.keySet.iterator().next
();
    }

    // helper function to make change on given key according to offset
    private void changeKey(String key, int offset) {
        int count = keyCountMap.get(key);
        keyCountMap.put(key, count + offset);
        Bucket curBucket = countBucketMap.get(count);
        Bucket newBucket;
        if (countBucketMap.containsKey(count + offset)) {
            // target Bucket already exists
            newBucket = countBucketMap.get(count + offset);
        } else {
            // add new Bucket
            newBucket = new Bucket(count + offset);
            countBucketMap.put(count + offset, newBucket);
            addBucketAfter(newBucket, offset == 1 ? curBucket : curBucket.pre);
        }
        newBucket.keySet.add(key);
        removeKeyFromBucket(curBucket, key);
    }

    private void removeKeyFromBucket(Bucket bucket, String key) {
        bucket.keySet.remove(key);
        if (bucket.keySet.size() == 0) {
            removeBucketFromList(bucket);
            countBucketMap.remove(bucket.count);
        }
    }

    private void removeBucketFromList(Bucket bucket) {
        bucket.pre.next = bucket.next;
        bucket.next.pre = bucket.pre;
```

```
        bucket.next = null;
        bucket.pre = null;
    }

    // add newBucket after preBucket
    private void addBucketAfter(Bucket newBucket, Bucket preBucket) {
        newBucket.pre = preBucket;
        newBucket.next = preBucket.next;
        preBucket.next.pre = newBucket;
        preBucket.next = newBucket;
    }
}
```

written by AaronLin1992 original link here

From Leetcoder.

Non-overlapping Intervals

Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Note:

1. You may assume the interval's end point is always bigger than its start point.
2. Intervals like [1,2] and [2,3] have borders "touching" but they don't overlap each other.

Example 1:

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

Output: 1

Explanation: [1,3] can be removed and the rest of intervals are non-overlapping.
```

Example 2:

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

Output: 2

Explanation: You need to remove two [1,2] to make the rest of intervals non-overlap
ping.
```

Example 3:

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

Output: 0

Explanation: You don't need to remove any of the intervals since they're already no
n-overlapping.
```

The answers will be available soon! Meanwhile you can go check out the answers in the discussion forum so far.

From Leetcoder.

Find Right Interval

Given a set of intervals, for each of the interval i, check if there exists an interval j whose start point is bigger than or equal to the end point of the interval i, which can be called that j is on the "right" of i.

For any interval i, you need to store the minimum interval j's index, which means that the interval j has the minimum start point to build the "right" relationship for interval i. If the interval j doesn't exist, store -1 for the interval i. Finally, you need output the stored value of each interval as an array.

Note:

1. You may assume the interval's end point is always bigger than its start point.
2. You may assume none of these intervals have the same start point.

Example 1:

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

Output: [-1]

Explanation: There is only one interval in the collection, so it outputs -1.
```

Example 2:

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

Output: [-1, 0, 1]

Explanation: There is no satisfied "right" interval for [3,4].
For [2,3], the interval [3,4] has minimum-"right" start point;
For [1,2], the interval [2,3] has minimum-"right" start point.
```

Example 3:

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

Output: [-1, 2, -1]

Explanation: There is no satisfied "right" interval for [1,4] and [3,4].
For [2,3], the interval [3,4] has minimum-"right" start point.
```

The answers will be available soon! Meanwhile you can go check out the answers in the discussion forum so far.

From Leetcoder.

Path Sum III

You are given a binary tree in which each node contains an integer value.

Find the number of paths that sum to a given value.

The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

The tree has no more than 1,000 nodes and the values are in the range -1,000,000 to 1,000,000.

Example:

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

Return 3. The paths that sum to 8 are:

1.  5 -> 3
2.  5 -> 2 -> 1
3. -3 -> 11
```

Solution 1

So the idea is similar as Two sum, using HashMap to store (key : the prefix sum, value : how many ways get to this prefix sum) , and whenever reach a node, we check if prefix sum - target exists in hashmap or not, if it does, we added up the ways of prefix sum - target into res.
For instance : in one path we have 1,2,-1,-1,2, then the prefix sum will be: 1, 3, 2, 1, 3, let's say we want to find target sum is 2, then we will have {2}, {1, 2, -1, -1, 2}, { 2, -1, -1, 2} ways.

I used global variable count, but obviously we can avoid global variable by passing the count from bottom up. The time complexity is O(n). This is my first post in discuss, open to any improvement or criticism. :)

```
    public int pathSum(TreeNode root, int sum) {
        HashMap<Integer, Integer> preSum = new HashMap();
        preSum.put(0,1);
        helper(root, 0, sum, preSum);
        return count;
    }
    int count = 0;
    public void helper(TreeNode root, int sum, int target, HashMap<Integer, Integ
er> preSum) {
        if (root == null) {
            return;
        }

        sum += root.val;

        if (preSum.containsKey(sum - target)) {
            count += preSum.get(sum - target);
        }

        if (!preSum.containsKey(sum)) {
            preSum.put(sum, 1);
        } else {
            preSum.put(sum, preSum.get(sum)+1);
        }

        helper(root.left, sum, target, preSum);
        helper(root.right, sum, target, preSum);
        preSum.put(sum, preSum.get(sum) - 1);
    }
```

Thanks for your advice, @StefanPochmann . Here is the modified version, concise and shorter:

```
    public int pathSum(TreeNode root, int sum) {
        HashMap<Integer, Integer> preSum = new HashMap();
        preSum.put(0,1);
        return helper(root, 0, sum, preSum);
    }

    public int helper(TreeNode root, int sum, int target, HashMap<Integer, Intege
r> preSum) {
        if (root == null) {
            return 0;
        }

        sum += root.val;
        int res = preSum.getOrDefault(sum - target, 0);
        preSum.put(sum, preSum.getOrDefault(sum, 0) + 1);

        res += helper(root.left, sum, target, preSum) + helper(root.right, sum, t
arget, preSum);
        preSum.put(sum, preSum.get(sum) - 1);
        return res;
    }
```

written by tankztc original link here

Solution 2

Each time find all the path start from current node
Then move start node to the child and repeat.
Time Complexity should be O(N^2) for the worst case and O(NlogN) for balanced binary Tree.

```
public class Solution {
    public int pathSum(TreeNode root, int sum) {
        if(root == null)
            return 0;
        return findPath(root, sum) + pathSum(root.left, sum) + pathSum(root.right
, sum);
    }

    public int findPath(TreeNode root, int sum){
        int res = 0;
        if(root == null)
            return res;
        if(sum == root.val)
            res++;
        res += findPath(root.left, sum - root.val);
        res += findPath(root.right, sum - root.val);
        return res;
    }

}
```

A better solution is suggested in 17ms O(n) java prefix sum by tankztc. It use a hash map to store all the prefix sum and each time check if the any subarray sum to the target, add with some comments:

```
    public int pathSum(TreeNode root, int sum) {
        Map<Integer, Integer> map = new HashMap<>();
        map.put(0, 1);  //Default sum = 0 has one count
        return backtrack(root, 0, sum, map);
    }
    //BackTrack one pass
    public int backtrack(TreeNode root, int sum, int target, Map<Integer, Integer
> map){
        if(root == null)
            return 0;
        sum += root.val;
        int res = map.getOrDefault(sum - target, 0);    //See if there is a subar
ray sum equals to target
        map.put(sum, map.getOrDefault(sum, 0)+1);
        //Extend to left and right child
        res += backtrack(root.left, sum, target, map) + backtrack(root.right, sum
, target, map);
        map.put(sum, map.get(sum)-1);   //Remove the current node so it wont affe
ct other path
        return res;
    }
```

written by star1993 original link here

Solution 3

For tree structure problems. recursion is usually intuitive and easy to write. lol

```
class Solution {
public:
    int pathSum(TreeNode* root, int sum) {
        if(!root) return 0;
        return sumUp(root, 0, sum) + pathSum(root->left, sum) + pathSum(root->rig
ht, sum);
    }
private:
    int sumUp(TreeNode* root, int pre, int& sum){
        if(!root) return 0;
        int current = pre + root->val;
        return (current == sum) + sumUp(root->left, current, sum) + sumUp(root->r
ight, current, sum);
    }
};
```

written by Undo original link here

From Leetcoder.

Find All Anagrams in a String

Given a string **s** and a **non-empty** string **p**, find all the start indices of **p**'s anagrams in **s**.

Strings consists of lowercase English letters only and the length of both strings**s** and **p** will not be larger than 20,100.

The order of output does not matter.

Example 1:

```
Input:
s: "cbaebabacd" p: "abc"

Output:
[0, 6]

Explanation:
The substring with start index = 0 is "cba", which is an anagram of "abc".
The substring with start index = 6 is "bac", which is an anagram of "abc".
```

Example 2:

```
Input:
s: "abab" p: "ab"

Output:
[0, 1, 2]

Explanation:
The substring with start index = 0 is "ab", which is an anagram of "ab".
The substring with start index = 1 is "ba", which is an anagram of "ab".
The substring with start index = 2 is "ab", which is an anagram of "ab".
```

Solution 1

Same idea from a fantastic sliding window template, please refer: https://discuss.leetcode.com/topic/30941/here-is-a-10-line-template-that-can-solve-most-substring-problems

Time Complexity will be O(n) because the "start" and "end" points will only move from left to right once.

```
public List<Integer> findAnagrams(String s, String p) {
    List<Integer> list = new ArrayList<>();
    if (s == null || s.length() == 0 || p == null || p.length() == 0) return list;
    int[] hash = new int[256]; //character hash
    //record each character in p to hash
    for (char c : p.toCharArray()) {
        hash[c]++;
    }
    //two points, initialize count to p's length
    int left = 0, right = 0, count = p.length();
    while (right < s.length()) {
        //move right everytime, if the character exists in p's hash, decrease the
count
        //current hash value >= 1 means the character is existing in p
        if (hash[s.charAt(right++)]-- >= 1) count--;

        //when the count is down to 0, means we found the right anagram
        //then add window's left to result list
        if (count == 0) list.add(left);

        //if we find the window's size equals to p, then we have to move left (nar
row the window) to find the new match window
        //++ to reset the hash because we kicked out the left
        //only increase the count if the character is in p
        //the count >= 0 indicate it was original in the hash, cuz it won't go bel
ow 0
        if (right - left == p.length() && hash[s.charAt(left++)]++ >= 0) count++;
    }
    return list;
}
```

written by NathanNi original link here

Solution 2

```
public List<Integer> findAnagrams(String s, String p) {
    int[] chars = new int[26];
    List<Integer> result = new ArrayList<>();

    if (s == null || p == null || s.length() < p.length())
        return result;
    for (char c : p.toCharArray())
        chars[c-'a']++;

    int start = 0, end = 0, count = p.length();
    // Go over the string
    while (end < s.length()) {
        // If the char at start appeared in p, we increase count
        if (end - start == p.length() && chars[s.charAt(start++)-'a']++ >= 0)
            count++;
        // If the char at end appeared in p (since it's not -1 after decreasing),
we decrease count
        if (--chars[s.charAt(end++)-'a'] >= 0)
            count--;
        if (count == 0)
            result.add(start);
    }

    return result;
}
```

written by dahui original link here

Solution 3

```java
    private List<Integer> findAnagrams(String s, String p) {
        List<Integer> res = new ArrayList<>();
        int[] win = new int[256];
        int[] pFixWin = new int[256];

        if (s.length() == 0 || p.length() == 0 || p.length() > s.length()) {
            return res;
        }

        /// pre - load the pWin
        for (int i = 0; i < p.length(); i++) {
            pFixWin[p.charAt(i)]++;
        }

        // pre-load the moving  window
        for (int i = 0; i < p.length(); i++) {
            win[s.charAt(i)]++;
        }

        for (int i = 0; i < s.length(); i++) {
            // for each position check if the numbers of each letter
            // coincide in the window and the fixed window
            if (isPContainedInS(pFixWin, win)) {
                res.add(i);
            }

            // evict from the window the char we just passed in S
            // add from the S window the next character
            if ((i + p.length()) < s.length()) {
                win[s.charAt(i)]--;
                win[s.charAt(i + p.length())]++;
            } else {
                break;
            }

        }
        return res;

    }

    private boolean isPContainedInS(int[] pFixWin, int[] win) {
        for (int i = 0; i < pFixWin.length; i++) {
            if (pFixWin[i] != win[i]) {
                return false;
            }
        }
        return true;
    }
```

written by federico original link here

Ternary Expression Parser

Given a string representing arbitrarily nested ternary expressions, calculate the result of the expression. You can always assume that the given expression is valid and only consists of digits `0-9`, `?`, `:`, `T` and `F` (`T` and `F` represent True and False respectively).

Note:

1. The length of the given string is ≤ 10000.
2. Each number will contain only one digit.
3. The conditional expressions group right-to-left (as usual in most languages).
4. The condition will always be either `T` or `F`. That is, the condition will never be a digit.
5. The result of the expression will always evaluate to either a digit `0-9`, `T` or `F`.

Example 1:

```
Input: "T?2:3"

Output: "2"

Explanation: If true, then result is 2; otherwise result is 3.
```

Example 2:

```
Input: "F?1:T?4:5"

Output: "4"

Explanation: The conditional expressions group right-to-left. Using parenthesis, it
 is read/evaluated as:

            "(F ? 1 : (T ? 4 : 5))"                  "(F ? 1 : (T ? 4 : 5))"
         -> "(F ? 1 : 4)"                 or       -> "(T ? 4 : 5)"
         -> "4"                                    -> "4"
```

Example 3:

```
Input: "T?T?F:5:3"

Output: "F"

Explanation: The conditional expressions group right-to-left. Using parenthesis, it
 is read/evaluated as:

            "(T ? (T ? F : 5) : 3)"                  "(T ? (T ? F : 5) : 3)"
         -> "(T ? F : 3)"                 or       -> "(T ? F : 5)"
         -> "F"                                    -> "F"
```

Solution 1

Iterate the expression from tail, whenever encounter a character before '?', calculate the right value and push back to stack.

P.S. this code is guaranteed only if "the given expression is valid" base on the requirement.

```
public String parseTernary(String expression) {
    if (expression == null || expression.length() == 0) return "";
    Deque<Character> stack = new LinkedList<>();

    for (int i = expression.length() - 1; i >= 0; i--) {
        char c = expression.charAt(i);
        if (!stack.isEmpty() && stack.peek() == '?') {

            stack.pop(); //pop '?'
            char first = stack.pop();
            stack.pop(); //pop ':'
            char second = stack.pop();

            if (c == 'T') stack.push(first);
            else stack.push(second);
        } else {
            stack.push(c);
        }
    }

    return String.valueOf(stack.peek());
}
```

written by NathanNi original link here

Solution 2

In order to pick out useful "?" and ":", we can always begin with**the last "?" and the first ":" after the chosen "?"**.

Therefore, directly seek for the last "?" (or you can simply put all "?" into a stack) and update the string depending on T or F until no more "?"s.

e.g.
"(F ? 1 : (T ? 4 : 5))" => "(F ? 1 : 4)" => "4"
"(T ? (T ? F : 5) : 3)" => "(T ? F : 3)" => "F"

EDIT:
Removed stack, added Java version.

Python:

```
class Solution(object):
    def parseTernary(self, expression):
        """
        :type expression: str
        :rtype: str
        """
        while len(expression) != 1:
            i = expression.rindex("?")    # begin with the last '?'.
            tmp = expression[i+1] if expression[i-1] == 'T' else expression[i+3]
            expression = expression[:i-1] + tmp + expression[i+4:]
        return expression
```

Java (It costs 7-lines):

```
public class Solution {
    public String parseTernary(String expression) {
        while (expression.length() != 1) {
            int i = expression.lastIndexOf("?");    // get the last shown '?'
            char tmp;
            if (expression.charAt(i-1) == 'T') { tmp = expression.charAt(i+1); }
            else { tmp = expression.charAt(i+3); }
            expression = expression.substring(0, i-1) + tmp + expression.substrin
g(i+4);
        }
        return expression;
    }
}
```

written by YJL1228 original link here

Solution 3

O(n) with stack

Collect chars from back to front on a stack, evaluate ternary sub-expressions as soon as possible:

```
def parseTernary(self, expression):
    stack = []
    for c in reversed(expression):
        stack.append(c)
        if stack[-2:-1] == ['?']:
            stack[-5:] = stack[-3 if stack[-1] == 'T' else -5]
    return stack[0]
```

Originally my check was `stack[-4::2] == [':', '?']`, but @YJL1228's is right, looking for `?` is enough.

O(n^2), several versions

Always evaluate/replace the last included ternary. So somewhat the same as the stack solution but only O(n^2). Didn't think of that and instead went right for the stack solution, but now that I saw it from others, I just had to write a few ways myself :-)

Version just working on the string:

```
def parseTernary(self, s):
    while len(s) > 1:
        i = s.rfind('?') - 1
        s = s[:i] + s[i+2 if s[i] == 'T' else i+4] + s[i+5:]
    return s
```

Version working on a list version of the string because it can be modified (need to reverse it because lists can only tell the first occurrence, not the last):

```
def parseTernary(self, s):
    a = list(s)[::-1]
    while len(a) > 1:
        i = a.index('?') - 3
        a[i:i+5] = a[i+2 if a[i+4] == 'T' else i]
    return a[0]
```

Version using a regular expression (reversing the string because `sub` doesn't support replacing the last occurrence but does support replacing the first):

```
def parseTernary(self, s):
    s = s[::-1]
    while len(s) > 1:
        s = re.sub('(.):(.)\?(.)', lambda m: m.group(1 + (m.group(3) == 'T')), s,
1)
    return s
```

written by StefanPochmann original link here

From Leetcoder.

K-th Smallest in Lexicographical Order

Given integers `n` and `k`, find the lexicographically k-th smallest integer in the range from `1` to `n`.

Note: $1 \le k \le n \le 10^9$.

Example:

```
Input:
n: 13   k: 2

Output:
10

Explanation:
The lexicographical order is [1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9], so the se
cond smallest number is 10.
```

Solution 1

This solution is inspired by @mgch's code.

General ideas:
1. Calculate the number of numbers that prefix with `result`.
Numbers prefixed by `result` are the union of the following intervals:
[`result` , `result` +1)
[`result` *10, (`result` +1)*10)
[`result` *100, (`result` +1)*100)
...
But they also shall belong to [1, n]
Therefore, we can easily calculate the number of elements prefixed by `result` using the following code (not considering overflow):

```
            int count = 0;
            for (int first = result, last = result + 1; // the first interval con
tains only one element: {result}
                first <= n; // the interval is not empty
                first *= 10, last *= 10) // increase a digit
            {
                // valid interval = [first, last) union [first, n]
                count += (min(n + 1, last) - first); // add the length of interva
l
            }
```

2. Search the next prefix, or search more detailedly.

- If the number of numbers that prefixed by `result` is smaller than the remaining `k`, we do not need to consider the numbers prefixed by `result` any more, and march to the next prefix: `result` +1;
- Otherwise, we need to search more detailedly, by appending another digit to the prefix. Such search shall start with `result` *10.

C++

```
class Solution {
public:
    int findKthNumber(int n, int k)
    {
        int result = 1;
        for(--k; k > 0; )
        {
            // calculate #|{result, result*, result**, result***, ...}|
            int count = 0;
            for (long long first = static_cast<long long>(result), last = first +
1;
                first <= n; // the interval is not empty
                first *= 10, last *= 10) // increase a digit
            {
                // valid interval = [first, last) union [first, n]
                count += static_cast<int>((min(n + 1LL, last) - first)); // add t
he length of interval
            }

            if (k >= count)
            {   // skip {result, result*, result**, result***, ...}
                // increase the current prefix
                ++result;
                k -= count;
            }
            else
            {   // not able to skip all of {result, result*, result**, result***,
...}
                // search more detailedly
                result *= 10;
                --k;
            }
        }
        return result;
    }
};
```

Python

```python
class Solution(object):
    def findKthNumber(self, n, k):
        result = 1;
        k -= 1
        while k > 0:
            count = 0
            interval = [result, result+1]
            while interval[0] <= n:
                count += (min(n+1, interval[1]) - interval[0])
                interval = [10*interval[0], 10*interval[1]]

            if k >= count:
                result += 1
                k -= count
            else:
                result *= 10
                k -= 1
        return result
```

Complexities:

- **Time Complexity:** O((log$n$) ^ 2)
  Here, log$n$ is the number of digits in n, and it is also the number of replications of appending zero to search detailedly. Each such appending introduces the increasement of prefix at most 10 times, and each increasement may require O(log$n$) time to calculate the number of numbers that prefixed by the `result`.
- **Space Complexity:** O(1)

written by zhiqing_xiao original link here

Solution 2

Original idea comes from
http://bookshadow.com/weblog/2016/10/24/leetcode-k-th-smallest-in-lexicographical-order/

Actually this is a denary tree (each node has 10 children). Find the kth element is to do a k steps preorder traverse of the tree.

Initially, image you are at node 1 (variable: curr),
the goal is move (k - 1) steps to the target node x. (substract steps from k after moving)
when k is down to 0, curr will be finally at node x, there you get the result.

we don't really need to do a exact k steps preorder traverse of the denary tree,**the idea is to calculate the steps between curr and curr + 1 (neighbor nodes in same level), in order to skip some unnecessary moves.**

Main function
Firstly, calculate how many steps curr need to move to curr + 1.

1. if the steps <= k, we know we can move to curr + 1, and narrow down k to k - steps.
2. else if the steps > k, that means the curr + 1 is actually behind the target node x in the preorder path, we can't jump to curr + 1. What we have to do is to move forward only 1 step (curr * 10 is always next preorder node) and repeat the iteration.

calSteps function

1. how to calculate the steps between curr and curr + 1?
   Here we come up a idea to calculate by level.
   Let n1 = curr, n2 = curr + 1.
   n2 is always the next right node beside n1's right most node (who shares the same ancestor "curr")
   (refer to the pic, 2 is right next to 1, 20 is right next to 19, 200 is right next to 199).
2. so, if n2 <= n, what means n1's right most node exists, we can simply add the number of nodes from n1 to n2 to steps.
3. else if n2 > n, what means n (the biggest node) is on the path between n1 to n2, add (n + 1 - n1) to steps.
4. organize this flow to "steps += Math.min(n + 1, n2) - n1; n1 *= 10; n2 *= 10;"

Here is the code snippet:

```java
public int findKthNumber(int n, int k) {
    int curr = 1;
    k = k - 1;
    while (k > 0) {
        int steps = calSteps(n, curr, curr + 1);
        if (steps <= k) {
            curr += 1;
            k -= steps;
        } else {
            curr *= 10;
            k -= 1;
        }
    }
    return curr;
}
//use long in case of overflow
public int calSteps(int n, long n1, long n2) {
    int steps = 0;
    while (n1 <= n) {
        steps += Math.min(n + 1, n2) - n1;
        n1 *= 10;
        n2 *= 10;
    }
    return steps;
}
```

written by NathanNi original link here

Solution 3

There is a good explanation at there.

The solution used DFS to search in a trie tree. The trick is to skip the sub-tree if the (current index + node number of the sub-tree) is smaller than the k.

The key problem is - How to count the nodes in a sub-tree?

The trie tree was made up of two kinds of sub-tree, the complete sub-tree with each node either has ten or zero children and the incomplete sub-tree with some inner nodes can has 1 to 9 children.

The complete tree's nodes number is easy to get, it will be something like 1, 11, 111, 1111...(each node has ten children plus the root)

The incomplete tree can be calculated like the flowing example:

for n = 213, the tree will be something like this:

```
                                        $
  /                 /                 /                 \
 1                 2                 3                 [4 ~ 9]

 /\                /\                /\                 /\
0, [1~9]         0,   1,[2~9]    0,[1~9]               ...
/      \          /   /
[0~9]  ...     [0~9] [0~3]
```

the sub-trees start with char '1' and '3'~'9' are complete trees, which '1' sub-tree with ranges in 1,10~19, 100~199 has 111 nodes and '3'~'9' each has 11 nodes (e.g. 3 with ranges in 3, 30~39).

the sub-tree starts with '2' is not full, i.e. the ranges are 2, 20~29, 200~213. But the '2' sub-tree still has a 'complete' tree like '3'~'9' with range in 2, 20~29, and we just need to add the remained leaf nodes which is (213%200)+1 or (213-200)+1.

The 200 is the most left node in the '2' sub-tree and 299 is the most right node in '2' sub-tree if it is complete.
it also showed the idea about how to judge a given sub-three is complete or incomplete:

We first **assume all sub-trees are complete** and test by following:

```
{
if (the most right node in a sub-tree is not greater than the number n)

    the sub-tree is complete with nodes number  = 111..(log10(n)+1 1s)

else if (the most left node in a sub-tree is greater than the number n)

    the sub-tree is complete with nodes number  = 11...  (log10(n) 1s)

else

    the sub-tree is incomplete with nodes number =
n - (the most left node in the subree) + 1 + 11... (log10(n) 1s)
}
```

The nodes number in a complete tree can be cached.

Then we just need to recursively decrease k by the nodes number of the skipped sub-tree until k reached 1.

Also be carefully the first layer is start from 1 while other layers are start from 0.

```
public class Solution {
    int countNum(int n){
        int i=0;
        while(n>0){
            n/=10;
            i++;
        }
        return i;
    }
    int getFullTreeNum(int depth){
        int sum=0, children=1;
        while(depth>0){
            sum+=children;
            children*=10;
            depth--;
        }
        return sum;
    }
    int getMax(int prefix, int depth){
        while(depth>0){
            prefix*=10;
            prefix+=9;
            depth--;
        }
        return prefix;
    }
    int getMin(int prefix, int depth){
        while(depth>0){
            prefix*=10;
            depth--;
        }
        return prefix;
    }
```

```
    int helper(int n, int k, int prefix, int depth){
        int lowNum=getFullTreeNum(depth), highNum=getFullTreeNum(depth-1);
        for(int i=(prefix==0?1:0);i<=9;i++){
            int nodeNum=0;
            if(getMax(prefix*10+i, depth-1)<=n){
                nodeNum=lowNum;
            }
            else if(getMin(prefix*10+i, depth-1)>n){
                nodeNum=highNum;
            }
            else{
                nodeNum=highNum+((n-getMin(prefix*10+i, depth-1))+1);
            }
            k-=nodeNum;
            if(k<=0){
                k+=nodeNum;
                if(k==1){
                    return prefix*10+i;
                }
                else {
                    return helper(n, k-1, prefix*10+i, depth-1);
                }
            }
        }
        return 0;
    }

    public int findKthNumber(int n, int k) {
        int depth=countNum(n);
        int index=0;
        return helper(n, k, 0, depth);
    }
}
```

written by pureklkl original link here

From Leetcoder.

Arranging Coins

You have a total of n coins that you want to form in a staircase shape, where every k-th row must have exactly k coins.

Given n, find the total number of **full** staircase rows that can be formed.

n is a non-negative integer and fits within the range of a 32-bit signed integer.

Example 1:

```
n = 5

The coins can form the following rows:
¤
¤ ¤
¤ ¤

Because the 3rd row is incomplete, we return 2.
```

Example 2:

```
n = 8

The coins can form the following rows:
¤
¤ ¤
¤ ¤ ¤
¤ ¤

Because the 4th row is incomplete, we return 3.
```

Solution 1

The idea is about quadratic equation, the formula to get the sum of arithmetic progression is
sum = (x + 1) * x / 2
so for this problem, if we know the the sum, then we can know the x = (-1 + sqrt(8 * n + 1)) / 2

```
public class Solution {
    public int arrangeCoins(int n) {
        return (int)((-1 + Math.sqrt(1 + 8 * (long)n)) / 2);
    }
}
```

written by mysun original link here

Solution 2

[JAVA] Clean Code with Explanations and Running Time [2 Solutions]

Full Solutions and Explanations

Solution 1

```
public class Solution {
    public int arrangeCoins(int n) {
        int start = 0;
        int end = n;
        int mid = 0;
        while (start <= end){
            mid = (start + end) >>> 1;
            if ((0.5 * mid * mid + 0.5 * mid ) <= n){
                start = mid + 1;
            }else{
                end = mid - 1;
            }
        }
        return start - 1;
    }
}
```

Complexity Analysis

Uniform cost model is used as Cost Model and `n` is the input number. `b` in this case would be `2`.

Time Complexity:

- Best Case `O(log_b(n))` : With respect to the input, the algorithm will always depend on the value of input.
- Average Case `O(log_b(n))` : With respect to the input, the algorithm will always depend on the value of input.
- Worst Case `O(log_b(n))` : With respect to the input, the algorithm will always depend on the value of input.

Auxiliary Space:

- Worst Case `O(1)` : Additional variables are of constant size.

Algorithm

Approach: Binary Search

The problem is basically asking the maximum length of consecutive number that has the running sum lesser or equal to `n`. In other word, find `x` that satisfy the following condition:

`1 + 2 + 3 + 4 + 5 + 6 + 7 + ... + x <= n`
`sum_{i=1}^x i <= n`
Running sum can be simplified,

`(x * ( x + 1)) / 2 <= n`
Binary search is used in this case to slowly narrow down the `x` that will satisfy the equation. Note that `0.5 * mid * mid + 0.5 * mid` does not have overflow issue as the intermediate result is implicitly autoboxed to `double` data type.

Solution 2

```
public class Solution {
    public int arrangeCoins(int n) {
        return (int) ((Math.sqrt(1 + 8.0 * n) - 1) / 2);
    }
}
```

Complexity Analysis

Uniform cost model is used as Cost Model and `n` is the input number. `b` in this case would be `2`.

Time Complexity:

- Best Case `O(1)` : With respect to the input, the algorithm will always perform basic mathematical operation that run in constant time.
- Average Case `O(1)` : With respect to the input, the algorithm will always perform basic mathematical operation that run in constant time.
- Worst Case `O(1)` : With respect to the input, the algorithm will always perform basic mathematical operation that run in constant time.

Auxiliary Space:

- Worst Case `O(1)` : No extra space is used.

Algorithm

Approach: Mathematics

The problem is basically asking the maximum length of consecutive number that has the running sum lesser or equal to `n`. In other word, find `x` that satisfy the following condition:

`1 + 2 + 3 + 4 + 5 + 6 + 7 + ... + x <= n`
`sum_{i=1}^x i <= n`
Running sum can be simplified,

`(x * ( x + 1)) / 2 <= n`
Using quadratic formula, `x` is evaluated to be,

`x = 1 / 2 * (-sqrt(8 * n + 1)-1)` (Inapplicable) or `x = 1 / 2 * (sqrt(8 * n + 1)-1)`
Negative root is ignored and positive root is used instead. Note that `8.0 * n` is very important because it will cause Java to implicitly autoboxed the intermediate result into `double` data type. The code will not work if it is simply `8 * n`. Alternatively, an explicit casting can be done `8 * (long) n)`.

---

written by ratchapong.t original link here

Solution 3

```
public int arrangeCoins(int n) {
        //convert int to long to prevent integer overflow
        long nLong = (long)n;

        long st = 0;
        long ed = nLong;

        long mid = 0;

        while (st <= ed){
            mid = st + (ed - st) / 2;

            if (mid * (mid + 1) <= 2 * nLong){
                st = mid + 1;
            }else{
                ed = mid - 1;
            }
        }

        return (int)(st - 1);
    }
```

written by larrywang2014 original link here

From Leetcoder.

Sequence Reconstruction

Check whether the original sequence `org` can be uniquely reconstructed from the sequences in `seqs`. The `org` sequence is a permutation of the integers from 1 to n, with $1 \le n \le 10^4$. Reconstruction means building a shortest common supersequence of the sequences in `seqs` (i.e., a shortest sequence so that all sequences in `seqs` are subsequences of it). Determine whether there is only one sequence that can be reconstructed from `seqs` and it is the `org` sequence.

Example 1:

```
Input:
org: [1,2,3], seqs: [[1,2],[1,3]]

Output:
false

Explanation:
[1,2,3] is not the only one sequence that can be reconstructed, because [1,3,2] is
also a valid sequence that can be reconstructed.
```

Example 2:

```
Input:
org: [1,2,3], seqs: [[1,2]]

Output:
false

Explanation:
The reconstructed sequence can only be [1,2].
```

Example 3:

```
Input:
org: [1,2,3], seqs: [[1,2],[1,3],[2,3]]

Output:
true

Explanation:
The sequences [1,2], [1,3], and [2,3] can uniquely reconstruct the original sequenc
e [1,2,3].
```

Example 4:

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

Output:
true
```

Solution 1

```java
public class Solution {
    public boolean sequenceReconstruction(int[] org, int[][] seqs) {
        Map<Integer, Set<Integer>> map = new HashMap<>();
        Map<Integer, Integer> indegree = new HashMap<>();

        for(int[] seq: seqs) {
            if(seq.length == 1) {
                if(!map.containsKey(seq[0])) {
                    map.put(seq[0], new HashSet<>());
                    indegree.put(seq[0], 0);
                }
            } else {
                for(int i = 0; i < seq.length - 1; i++) {
                    if(!map.containsKey(seq[i])) {
                        map.put(seq[i], new HashSet<>());
                        indegree.put(seq[i], 0);
                    }

                    if(!map.containsKey(seq[i + 1])) {
                        map.put(seq[i + 1], new HashSet<>());
                        indegree.put(seq[i + 1], 0);
                    }

                    if(map.get(seq[i]).add(seq[i + 1])) {
                        indegree.put(seq[i + 1], indegree.get(seq[i + 1]) + 1);
                    }
                }
            }
        }

        Queue<Integer> queue = new LinkedList<>();
        for(Map.Entry<Integer, Integer> entry: indegree.entrySet()) {
            if(entry.getValue() == 0) queue.offer(entry.getKey());
        }

        int index = 0;
        while(!queue.isEmpty()) {
            int size = queue.size();
            if(size > 1) return false;
            int curr = queue.poll();
            if(index == org.length || curr != org[index++]) return false;
            for(int next: map.get(curr)) {
                indegree.put(next, indegree.get(next) - 1);
                if(indegree.get(next) == 0) queue.offer(next);
            }
        }
        return index == org.length && index == map.size();
    }
}
```

written by xieta00221 original link here

Solution 2

For `org` to be uniquely reconstructible from `seqs` we need to satisfy 2 conditions:

1. Every sequence in `seqs` should be a subsequence in `org`. This part is obvious.
2. Every 2 consecutive elements in `org` should be consecutive elements in some sequence from `seqs`. Why is that? Well, suppose condition 1 is satisfied. Then for 2 any consecutive elements `x` and `y` in `org` we have 2 options.
   - We have both `x` and `y` in some sequence from `seqs`. Then (as condition 1 is satisfied) they must be consequtive elements in this sequence.
   - There is no sequence in `seqs` that contains both `x` and `y`. In this case we cannot uniquely reconstruct `org` from `seqs` as sequence with `x` and `y` switched would also be a valid original sequence for `seqs`.

So this are 2 necessary criterions. It is pretty easy to see that this are also sufficient criterions for `org` to be uniquely reconstructible (there is only 1 way to reconstruct sequence when we know that condition 2 is satisfied).

To implement this idea I have `idxs` hash that maps item to its index in `org` sequence to check condition 1. And I have `pairs` set that holds all consequitive element pairs for sequences from `seqs` to check condition 2 (I also consider first elements to be paired with previous `undefined` elements, it is necessary to check this).

```
var sequenceReconstruction = function(org, seqs) {
    const pairs = {};
    const idxs = {};

    for (let i = 0; i < org.length; i++)
        idxs[org[i]] = i;

    for (let j = 0; j < seqs.length; j++) {
        const s = seqs[j];
        for (let i = 0; i < s.length; i++) {
            if (idxs[s[i]] == null)
                return false;
            if (i > 0 && idxs[s[i - 1]] >= idxs[s[i]])
                return false;
            pairs[`${s[i - 1]}_${s[i]}`] = 1;
        }
    }

    for (let i = 0; i < org.length; i++)
        if (pairs[`${org[i - 1]}_${org[i]}`] == null)
            return false;

    return true;
};
```

written by dettier original link here

Solution 3

The basic idea is to count how many numbers are smaller(self include) than the current number.
We then compare this count to the org.
It is pretty like the idea of count sort.

```
public class Solution {
    public boolean sequenceReconstruction(int[] org, int[][] seqs) {
        int len = org.length;
        int[] map = new int[len + 1];//map number to its index
        Arrays.fill(map, -1);
        int[] memo = new int[org.length];//count how many numbers are smaller(on
the right)
        for (int i = 0; i < len; i++) {
            map[org[i]] = i;
        }
        for (int[] seq : seqs) {
            if (seq.length == 0) continue;
            int prev = seq[0];
            if (prev <= 0 || prev > len || map[prev] == -1) return false;
            for (int i = 1; i < seq.length; i++) {
                int curr = seq[i];
                if (curr <= 0 || curr > len || map[curr] == -1) return false;
                memo[map[prev]] = Math.max(memo[map[prev]], len - map[curr] + 1);
                prev = curr;
            }
            memo[map[prev]] = Math.max(memo[map[prev]], 1);
        }
        for (int i = 0; i < memo.length; i++) {
            if (memo[i] != len - i) return false;
        }
        return true;
    }
}
```

written by jianhao2 original link here

From Leetcoder.

Arithmetic Slices II - Subsequence

A sequence of numbers is called arithmetic if it consists of at least three elements and if the difference between any two consecutive elements is the same.

For example, these are arithmetic sequences:

```
1, 3, 5, 7, 9
7, 7, 7, 7
3, -1, -5, -9
```

The following sequence is not arithmetic.

```
1, 1, 2, 5, 7
```

A zero-indexed array A consisting of N numbers is given. A **subsequence** slice of that array is any sequence of integers (P_0, P_1, ..., P_k) such that 0 ≤ P_0 1 k

A **subsequence** slice (P_0, P_1, ..., P_k) of array A is called arithmetic if the sequence A[P_0], A[P_1], ..., A[P_{k-1}], A[P_k] is arithmetic. In particular, this means that k ≥ 2.

The function should return the number of arithmetic subsequence slices in the array A.

The input contains N integers. Every integer is in the range of -2^{31} and $2^{31}-1$ and 0 ≤ N ≤ 1000. The output is guaranteed to be less than $2^{31}-1$.

Example:

```
Input: [2, 4, 6, 8, 10]

Output: 7

Explanation:
All arithmetic subsequence slices are:
[2,4,6]
[4,6,8]
[6,8,10]
[2,4,6,8]
[4,6,8,10]
[2,4,6,8,10]
[2,6,10]
```

Solution 1

```
    public int numberOfArithmeticSlices(int[] A) {
        int re = 0;
        HashMap<Integer, Integer>[] maps = new HashMap[A.length];
        for(int i=0; i<A.length; i++) {
            maps[i] = new HashMap<>();
            int num = A[i];
            for(int j=0; j<i; j++) {
                if((long)num-A[j]>Integer.MAX_VALUE) continue;
                if((long)num-A[j]<Integer.MIN_VALUE) continue;
                int diff = num - A[j];
                int count = maps[j].getOrDefault(diff, 0);
                maps[i].put(diff, maps[i].getOrDefault(diff,0)+count+1);
                re += count;
            }
        }
        return re;
    }
```

written by aaaeeeo original link here

Solution 2

First of all, the standard O(N^2) DP solution written with C++ goes MLE/TLE in LC, but works pretty well when written with JAVA/Python :(
It looks like this (Python version):

```
class Solution(object):
    def numberOfArithmeticSlices(self, A):
        dp = [defaultdict(int) for i in range(len(A))]
        res = 0
        for i in range(1, len(A)):
            for j in range(i):
                step = A[i]-A[j]
                dp[i][step] += 1
                if step in dp[j]:
                    dp[i][step]+= dp[j][step]
                    res += dp[j][step]
        return res
```

Obviously, it maintains an array of dictionary to store the number of arithmetic subsequences (including length 2) ending with A[i]. As a result, the time and space complexity are both O(N^2) which I think is quite reasonable to deal with 0 ≤ N ≤ 1000 in LC. But it fails in C++. So I tweak the meaning of DP equation from:

```
DP[i][d] = the number of arithmetic subsequences ending with A[i], difference is
d. (NOTE here the length of valid subsequences can be 2)
```

to

```
DP[i][d] = the number of arithmetic subsequences whose last but one number is A[i
], difference is d.
```

After that, the length of valid subsequences we record must be at least 3. It does save memory and finally passes LC in 423 ms:

```
class Solution {
public:
    int numberOfArithmeticSlices(vector<int>& A) {
        if (A.empty()) return 0;
        int n = A.size();
        vector<unordered_map<long long, int >> dp(n);
        unordered_set<int> s(A.begin(), A.end());
        int res = 0;
        for (int i = 1; i < n; ++i) {
            for (int j = i-1; j >= 0; --j) {
                long long d = (long long)A[i] - (long long)A[j];
                int tmp = dp[j].count(d) ? dp[j][d] : 0;
                if (tmp) res += tmp;
                if (s.count(A[i]+d)) dp[i][d] += 1 + tmp;
            }
        }
        return res;
    }
};
```

written by vesion original link here

Solution 3

```
public int numberOfArithmeticSlices(int[] A) {
        if(A==null||A.length<3) return 0;
        List<Map<Integer,Integer>> list=new ArrayList<Map<Integer,Integer>>();
        int res=0;
        for(int i=1;i<A.length;i++){
            Map<Integer,Integer> map=new HashMap<Integer,Integer>();
            for(int j=0;j<i;j++){
                if((long)A[i]-(long)A[j]>Integer.MAX_VALUE) continue;
                if((long)A[i]-(long)A[j]<Integer.MIN_VALUE) continue;
                int dif=A[i]-A[j];
                if(j==0){
                    map.put(dif,1);
                    continue;
                }
                Map<Integer,Integer> temp=list.get(j-1);
                int sum=0;
                if(temp.containsKey(dif)){
                    sum=temp.get(dif);
                }
                if(map.containsKey(dif)){
                    map.put(dif,map.get(dif)+sum+1);
                }else{
                    map.put(dif,sum+1);
                }
                res+=sum;
            }
            list.add(map);
        }
        return res;
    }
```

It seems that there is no better way to solve this problem using O(N^2) space and time. If you have better algorithm, please share your thoughts or solution, thanks!

I have viewed all four posts(at least there are four posts when I am still busy typing mine), and everyone seems to have the same issue facing TLE or MLE or both. Well, for me, I have encountered both.

Since nobody has posted java solution, I will share my ways of tackling this problem.

I will skip the logic and algorithms here, since everybody is doing it the same way, using DP. For each element, store all possible difference and its number of arithmetic sequences(including 2 element sequence).

At first, I was using Map<Long,Integer> instead of Map<Integer,Integer> considering subtraction of two integers could result in overflow. But later I realized if the result is bigger than the max value of integer or smaller than the min value of integer, there is no way to have a valid a 3rd integer element to form a arithmetic sequence. So by adding two if()*continue, and thus replacing Map<Long,Integer> to Map<Integer,Integer> it can save both time and memory. After doing this, my code can pass the judge about 4 out of 5 times, which means it can still have TLE or MLE

sometimes. Then I modified the outer loop's parameter i to start from 1 instead of 0 and added if(j==0) statement to do less map operations for every j==0 case. And now it goes through every time! (Well, by saying every time, I mean I submitted 5 times in a row and they all went through.)

I believe the first change is the main cause to help my code get through, and the second change is just the minor. Hope these can give you guys some hints and help, please do share your plans of reducing either time or space here if you have.
Happy coding!

written by KnightY original link here

From Leetcoder.

Number of Boomerangs

Given n points in the plane that are all pairwise distinct, a "boomerang" is a tuple of points `(i, j, k)` such that the distance between `i` and `j` equals the distance between `i` and `k` (**the order of the tuple matters**).

Find the number of boomerangs. You may assume that n will be at most **500** and coordinates of points are all in the range **[-10000, 10000]** (inclusive).

Example:

```
Input:
[[0,0],[1,0],[2,0]]

Output:
2

Explanation:
The two boomerangs are [[1,0],[0,0],[2,0]] and [[1,0],[2,0],[0,0]]
```

Solution 1

```
public int numberOfBoomerangs(int[][] points) {
    int res = 0;
    Map<Integer, Integer> map = new HashMap<>();

    for(int i=0; i<points.length; i++) {
        for(int j=0; j<points.length; j++) {
            if(i == j)
                continue;

            int d = getDistance(points[i], points[j]);
            map.put(d, map.getOrDefault(d, 0) + 1);
        }

        for(int val : map.values()) {
            res += val * (val-1);
        }
        map.clear();
    }

    return res;
}

private int getDistance(int[] a, int[] b) {
    int dx = a[0] - b[0];
    int dy = a[1] - b[1];

    return dx*dx + dy*dy;
}

Time complexity:  O(n^2)
Space complexity: O(n)
```

written by asurana28 original link here

Solution 2

The idea is simple, for every point, we aggregate points with the same distance and put them in a `Set`.

Update:

I got this solution during the contest, and as troy351 said, Actually we don't need `Map<Integer, Set>`, and `Map<Integer, Integer>` is enough.

```
public class Solution {
    public int numberOfBoomerangs(int[][] points) {
        if(points.length==0 || points[0].length==0) return 0;
        int ret = 0;
        for(int i=0;i<points.length;i++){
            Map<Integer, Set<int[]>> map = new HashMap<>();
            int[] p = points[i];
            for(int j=0;j<points.length;j++){
                if(j==i) continue;
                int[] q = points[j];
                int dis = getDis(p, q);
                if(!map.containsKey(dis)) map.put(dis, new HashSet<int[]>());
                map.get(dis).add(q);
            }
            for(Integer key : map.keySet()){
                int size = map.get(key).size();
                if(size>=2) ret += (size*(size-1));
            }
        }
        return ret;
    }
    public int getDis(int[] p, int[] q){
        int a = p[0]-q[0];
        int b = p[1]-q[1];
        return a*a+b*b;
    }
}
```

written by YuTingLiu original link here

Solution 3

For each point `i`, `map<` distance `d,` count of all points at distance `d` from `i>`. Given that count, choose `2` (with permutation) from it, to form a boomerang with point `i`.
[use `long` appropriately for `dx`, `dy` and `key`; though not required for the given test cases]

Time Complexity: `O(n^2)`

Updated: Using initial size for the map to avoid table resizing. Thanks @StefanPochmann

```
int numberOfBoomerangs(vector<pair<int, int>>& points) {

    int res = 0;

    // iterate over all the points
    for (int i = 0; i < points.size(); ++i) {

        unordered_map<long, int> group(points.size());

        // iterate over all points other than points[i]
        for (int j = 0; j < points.size(); ++j) {

            if (j == i) continue;

            int dy = points[i].second - points[j].second;
            int dx = points[i].first - points[j].first;

            // compute squared euclidean distance from points[i]
            int key = dy * dy;
            key += dx * dx;

            // accumulate # of such "j"s that are "key" distance from "i"
            ++group[key];
        }

        for (auto& p : group) {
            if (p.second > 1) {
                /*
                 * for all the groups of points,
                 * number of ways to select 2 from n =
                 * nP2 = n!/(n - 2)! = n * (n - 1)
                 */
                res += p.second * (p.second - 1);
            }
        }
    }

    return res;
}
```

written by kdtree original link here

From Leetcoder.

Minimum Number of Arrows to Burst Balloons

There are a number of spherical balloons spread in two-dimensional space. For each balloon, provided input is the start and end coordinates of the horizontal diameter. Since it's horizontal, y-coordinates don't matter and hence the x-coordinates of start and end of the diameter suffice. Start is always smaller than end. There will be at most 10^4 balloons.

An arrow can be shot up exactly vertically from different points along the x-axis. A balloon with x_{start} and x_{end} bursts by an arrow shot at x if $x_{start} \le x \le x_{end}$. There is no limit to the number of arrows that can be shot. An arrow once shot keeps travelling up infinitely. The problem is to find the minimum number of arrows that must be shot to burst all balloons.

Example:

```
Input:
[[10,16], [2,8], [1,6], [7,12]]

Output:
2

Explanation:
One way is to shoot one arrow for example at x = 6 (bursting the balloons [2,8] and
 [1,6]) and another arrow at x = 11 (bursting the other two balloons).
```

Solution 1

First, we sort balloons by increasing points.end (if ends are the same, then by increasing of points.start). Every time arrow shot points.end, say, points[i].second. If next balloon.start <= points[i].second, it is also shot, thus we continue.

```
    int findMinArrowShots(vector<pair<int, int>>& points) {
        int count = 0, arrow = INT_MIN;
        sort(points.begin(), points.end(), mysort);
        for(int i = 0; i<points.size(); i++){
            if(arrow!=INT_MIN && points[i].first<=arrow){continue;} //former arro
w shot points[i]
            arrow = points[i].second; // new arrow shot the end of points[i]
            count++;
        }
        return count;
    }
    static bool mysort(pair<int, int>& a, pair<int, int>& b){
        return a.second==b.second?a.first<b.first:a.second<b.second;
    }
```

written by westwatermelon original link here

Solution 2

```
public int findMinArrowShots(int[][] points) {
 if(points==null || points.length==0 || points[0].length==0) return 0;
 Arrays.sort(points, new Comparator<int[]>() {
  public int compare(int[] a, int[] b) {
   if(a[0]==b[0]) return a[1]-b[1];
   else return a[0]-b[0];
  }
 });

 int minArrows = 1;
 int arrowLimit = points[0][1];
 for(int i=1;i<points.length;i++) {
  int[] baloon = points[i];
  if(baloon[0]<=arrowLimit) {
   arrowLimit=Math.min(arrowLimit, baloon[1]);
  } else {
   minArrows++;
   arrowLimit=baloon[1];
  }
 }
 return minArrows;
}
```

written by fabrizio3 original link here

Solution 3

```
public class Solution {
    public int findMinArrowShots(int[][] points) {
        if(points == null || points.length < 1) return 0;
        Arrays.sort(points, (a, b)->(a[0]-b[0]));
        int result = 1;
        int end = points[0][1];
        for(int i = 1; i < points.length; i ++) {
            if(points[i][0] > end) {
                result ++;
                end = points[i][1];
            } else {
                end = Math.min(end, points[i][1]);
            }
        }
        return result;
    }
}
```

written by yuejn93 original link here

From Leetcoder.

Minimum Moves to Equal Array Elements

Given a **non-empty** integer array of size n, find the minimum number of moves required to make all array elements equal, where a move is incrementing $n - 1$ elements by 1.

Example:

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

Output:
3

Explanation:
Only three moves are needed (remember each move increments two elements):

[1,2,3]  =>  [2,3,3]  =>  [3,4,3]  =>  [4,4,4]
```

Solution 1

Add `1` to `n - 1` elements is the same as subtracting `1` from one element, w.r.t goal of making the elements in the array equal.
So, best way to do this is make all the elements in the array equal to the `min` element.
`sum(array) - n * minimum`

```
public class Solution {
    public int minMoves(int[] nums) {
        if (nums.length == 0) return 0;
        int min = nums[0];
        for (int n : nums) min = Math.min(min, n);
        int res = 0;
        for (int n : nums) res += n - min;
        return res;
    }
}
```

written by kdtree original link here

Solution 2

Incrementing all but one is equivalent to decrementing that one. So let's do that instead. How many single-element decrements to make all equal? No point to decrementing below the current minimum, so how many single-element decrements to make all equal to the current minimum? Just take the difference from what's currently there (the sum) to what we want (n times the minimum).

Python:

```
def minMoves(self, nums):
    return sum(nums) - len(nums) * min(nums)
```

Ruby:

```
def min_moves(nums)
  nums.inject(:+) - nums.size * nums.min
end
```

Java (ugh :-):

```
public int minMoves(int[] nums) {
    return IntStream.of(nums).sum() - nums.length * IntStream.of(nums).min().getA
sInt();
}
```

C++ (more ugh):

```
int minMoves(vector<int>& nums) {
    return accumulate(begin(nums), end(nums), 0) - nums.size() * *min_element(beg
in(nums), end(nums));
}
```

written by StefanPochmann original link here

Solution 3

let's define sum as the sum of all the numbers, before any moves; minNum as the min number int the list; n is the length of the list;

After, say m moves, we get all the numbers as x , and we will get the following equation

```
sum + m * (n - 1) = x * n
```

and actually,

```
x = minNum + m
```

and finally, we will get

```
sum - minNum * n = m
```

So, it is clear and easy now.

written by wang.senyuan original link here

From Leetcoder.

4Sum II

Given four lists A, B, C, D of integer values, compute how many tuples `(i, j, k, l)` there are such that `A[i] + B[j] + C[k] + D[l]` is zero.

To make problem a bit easier, all A, B, C, D have same length of N where $0 \le N \le 500$. All integers are in the range of -2^{28} to $2^{28} - 1$ and the result is guaranteed to be at most $2^{31} - 1$.

Example:

```
Input:
A = [ 1, 2]
B = [-2,-1]
C = [-1, 2]
D = [ 0, 2]

Output:
2

Explanation:
The two tuples are:
1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0
2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0
```

Solution 1

```
public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {
    Map<Integer, Integer> map = new HashMap<>();

    for(int i=0; i<C.length; i++) {
        for(int j=0; j<D.length; j++) {
            int sum = C[i] + D[j];
            map.put(sum, map.getOrDefault(sum, 0) + 1);
        }
    }

    int res=0;
    for(int i=0; i<A.length; i++) {
        for(int j=0; j<B.length; j++) {
            res += map.getOrDefault(-1 * (A[i]+B[j]), 0);
        }
    }

    return res;
}

Time complexity:  O(n^2)
Space complexity: O(n^2)
```

written by asurana28 original link here

Solution 2

```
def fourSumCount(self, A, B, C, D):
    AB = collections.Counter(a+b for a in A for b in B)
    return sum(AB[-c-d] for c in C for d in D)
```

written by StefanPochmann original link here

Solution 3

```
public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {
 Map<Integer,Integer> sums = new HashMap<>();
 int count = 0;
 for(int i=0; i<A.length;i++) {
  for(int j=0;j<B.length;j++){
   int sum = A[i]+B[j];
   if(sums.containsKey(sum)) {
    sums.put(sum, sums.get(sum)+1);
   } else {
    sums.put(sum, 1);
   }
  }
 }
 for(int k=0; k<A.length;k++) {
  for(int z=0;z<C.length;z++){
   int sum = -(C[k]+D[z]);
   if(sums.containsKey(sum)) {
    count+=sums.get(sum);
   }
  }
 }
 return count;
}
```

Take the arrays A and B, and compute all the possible sums of two elements. Put the sum in the Hash map, and increase the hash map value if more than 1 pair sums to the same value.

Compute all the possible sums of the arrays C and D. If the hash map contains the opposite value of the current sum, increase the count of four elements sum to 0 by the counter in the map.

written by fabrizio3 original link here

From Leetcoder.

Assign Cookies

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie. Each child i has a greed factor g_i, which is the minimum size of a cookie that the child will be content with; and each cookie j has a size s_j. If $s_j >= g_i$, we can assign the cookie j to the child i, and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number.

Note:
You may assume the greed factor is always positive.
You cannot assign more than one cookie to one child.

Example 1:

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

Output: 1

Explanation: You have 3 children and 2 cookies. The greed factors of 3 children are
 1, 2, 3.
And even though you have 2 cookies, since their size is both 1, you could only make
 the child whose greed factor is 1 content.
You need to output 1.
```

Example 2:

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

Output: 2

Explanation: You have 2 children and 3 cookies. The greed factors of 2 children are
 1, 2.
You have 3 cookies and their sizes are big enough to gratify all of the children,
You need to output 2.
```

Solution 1

```
Arrays.sort(g);
Arrays.sort(s);
int i = 0;
for(int j=0;i<g.length && j<s.length;j++) {
 if(g[i]<=s[j]) i++;
}
return i;
```

Just assign the cookies starting from the child with less greediness to maximize the number of happy children .

written by fabrizio3 original link here

Solution 2

My solution from the contest:

```
def findContentChildren(self, g, s):
    g.sort()
    s.sort()
    res = 0
    i = 0
    for e in s:
        if i == len(g):
            break
        if e >= g[i]:
            res += 1
            i += 1
    return res
```

O(nlogn) time and O(1) space

written by dalwise original link here

Solution 3

```
class Solution(object):
    def findContentChildren(self, g, s):
        """
        :type g: List[int]
        :type s: List[int]
        :rtype: int
        """
        g.sort()
        s.sort()

        childi = 0
        cookiei = 0

        while cookiei < len(s) and childi < len(g):
            if s[cookiei] >= g[childi]:
                childi += 1
            cookiei += 1

        return childi
```

written by vikram4 original link here

From Leetcoder.

132 Pattern

Given a sequence of n integers $a_1, a_2, \ldots, a_n$, a 132 pattern is a subsequence a_i, a_j, a_k such that **i** j k and a_i **k j**. Design an algorithm that takes a list of n numbers as input and checks whether there is a 132 pattern in the list.

Note: n will be less than 15,000.

Example 1:

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

Output: False

Explanation: There is no 132 pattern in the sequence.
```

Example 2:

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

Output: True

Explanation: There is a 132 pattern in the sequence: [1, 4, 2].
```

Example 3:

```
Input: [-1, 3, 2, 0]

Output: True

Explanation: There are three 132 patterns in the sequence: [-1, 3, 2], [-1, 3, 0] a
nd [-1, 2, 0].
```

Solution 1

We want to search for a subsequence (`s1,s2,s3`)

INTUITION: The problem would be simpler if we want to find sequence with `s1 > s2 > s3`, we just need to find `s1`, followed by `s2` and `s3`. Now if we want to find a 132 sequence, we need to switch up the order of searching. we want to first find `s2`, followed by `s3`, then `s1`.

IDEA: We can start from either side but I think starting from the end allow us to finish in a single pass. The idea is to start from end and search for a candidate for `s2` and `s3`. A number becomes a candidate for `s3` if there is any number on the left of `s2` that is bigger than it.

DETECTION: Keep track of the largest candidate of `s3` and once we encounter any number smaller than `s3`, we know we found a valid sequence since `s1 < s3` implies `s1 < s2`.

IMPLEMENTATION:

1. Have a `stack`, each time we store a new number, we first `pop` out all numbers that are smaller than that number. The numbers that are `popped` out becomes candidate for `s3`.
2. We keep track of the `maximum` of such `s3` (which is always the most recently `popped` number from the `stack`).
3. Once we encounter any number smaller than `s3`, we know we found a valid sequence since `s1 < s3` implies `s1 < s2`.

RUNTIME: Each item is `pushed` and `popped` once at most, the time complexity is therefore O(n).

EXAMPLE:
`i = 6`, **nums** = [9, 11, 8, 9, 10, 7, `9`], **S1 candidate** = `9`, **S3 candidate** = `None`, **Stack** = `Empty`
`i = 5`, **nums** = [9, 11, 8, 9, 10, `7`, 9], **S1 candidate** = `7`, **S3 candidate** = `None`, **Stack** = `[9]`
`i = 4`, **nums** = [9, 11, 8, 9, `10`, 7, 9], **S1 candidate** = `10`, **S3 candidate** = `None`, **Stack** = `[9,7]`
`i = 3`, **nums** = [9, 11, 8, `9`, 10, 7, 9], **S1 candidate** = `9`, **S3 candidate** = `9`, **Stack** = `[10]`
`i = 2`, **nums** = [9, 11, `8`, 9, 10, 7, 9], **S1 candidate** = `8`, **S3 candidate** = `9`, **Stack** = `[10,9]` **We have `8<9`, sequence found!**

EDIT: Thanks @Pumpkin78 and @dalwise for pointing out that the maximum candidate for s3 is always the recently popped number from the stack, because if we encounter any entry smaller than the current candidate, the function would already have returned.

```
    bool find132pattern(vector<int>& nums) {
        int s3 = INT_MIN;
        stack<int> st;
        for( int i = nums.size()-1; i >= 0; i -- ){
            if( nums[i] < s3 ) return true;
            else while( !st.empty() && nums[i] > st.top() ){
              s3 = st.top(); st.pop();
            }
            st.push(nums[i]);
        }
        return false;
    }
```

written by jade86 original link here

Solution 2

The idea is that we can use a stack to keep track of previous min-max intervals.

Here is the principle to maintain the stack:

For each number `num` in the array

If stack is empty:

- push a new Pair of `num` into stack

If stack is not empty:

- if `num` < `stack.peek().min`, push a new Pair of `num` into stack
- if `num` >= `stack.peek().min`, we first pop() out the peek element, denoted as `last`
  - if `num` < `last.max`, we are done, return `true`;
  - if `num` >= `last.max`, we merge `num` into `last`, which means `last.max` = `num`.
    Once we update `last`, if stack is empty, we just push back `last`.
    However, the crucial part is:
    If stack is not empty, the updated `last` might:
    - Entirely covered stack.peek(), i.e. `last.min` < `stack.peek().min` (which is always true) && `last.max` >= `stack.peek().max`, in which case we keep popping out stack.peek().
    - Form a 1-3-2 pattern, we are done ,return `true`

So at any time in the stack, **non-overlapping** `Pairs` are formed in descending order by their min value, which means the min value of peek element in the stack is always the min value globally.

```
   class Pair{
        int min, max;
        public Pair(int min, int max){
            this.min = min;
            this.max = max;
        }
    }
    public boolean find132pattern(int[] nums) {
        Stack<Pair> stack = new Stack();
        for(int n: nums){
            if(stack.isEmpty() || n <stack.peek().min ) stack.push(new Pair(n,n))
;
            else if(n > stack.peek().min){
                Pair last = stack.pop();
                if(n < last.max) return true;
                else {
                    last.max = n;
                    while(!stack.isEmpty() && n >= stack.peek().max) stack.pop();
                    // At this time, n < stack.peek().max (if stack not empty)
                    if(!stack.isEmpty() && stack.peek().min < n) return true;
                    stack.push(last);
                }

            }
        }
        return false;
    }
```

written by leogogogo original link here

Solution 3

Idea: Find peak and bottom

For every [bottom, peak], find if there is one number `bottom<number<peak`.

```
public class Solution {
    public boolean find132pattern(int[] nums) {
        if(nums.length<3) return false;
        Integer low = null, high = null;
        int start = 0, end = 0;
        while(start<nums.length-1){
            while(start<nums.length-1 && nums[start]>=nums[start+1]) start++;
            // start is lowest now
            int m = start+1;
            while(m<nums.length-1 && nums[m]<=nums[m+1]) m++;
            // m is highest now
            int j = m+1;
            while(j<nums.length){
                if(nums[j]>nums[start] && nums[j]<nums[m]) return true;
                j++;
            }
            start = m+1;
        }
        return false;
    }
}
```

written by YuTingLiu original link here

From Leetcoder.

Repeated Substring Pattern

Given a non-empty string check if it can be constructed by taking a substring of it and appending multiple copies of the substring together. You may assume the given string consists of lowercase English letters only and its length will not exceed 10000.

Example 1:

```
Input: "abab"

Output: True

Explanation: It's the substring "ab" twice.
```

Example 2:

```
Input: "aba"

Output: False
```

Example 3:

```
Input: "abcabcabcabc"

Output: True

Explanation: It's the substring "abc" four times. (And the substring "abcabc" twice
.)
```

Solution 1

```java
public boolean repeatedSubstringPattern(String str) {
 int l = str.length();
 for(int i=l/2;i>=1;i--) {
  if(l%i==0) {
   int m = l/i;
   String subS = str.substring(0,i);
   StringBuilder sb = new StringBuilder();
   for(int j=0;j<m;j++) {
    sb.append(subS);
   }
   if(sb.toString().equals(str)) return true;
  }
 }
 return false;
}
```

1. The length of the repeating substring must be a divisor of the length of the input string
2. Search for all possible divisor of `str.length`, starting for `length/2`
3. If `i` is a divisor of `length`, repeat the substring from `0` to `i` the number of times `i` is contained in `s.length`
4. If the repeated substring is equals to the input `str` return `true`

written by fabrizio3 original link here

Solution 2

```
public boolean repeatedSubstringPattern(String str) {
        //This is the kmp issue
        int[] prefix = kmp(str);
        int len = prefix[str.length()-1];
        int n = str.length();
        return (len > 0 && n%(n-len) == 0);
    }
    private int[] kmp(String s){
        int len = s.length();
        int[] res = new int[len];
        char[] ch = s.toCharArray();
        int i = 0, j = 1;
        res[0] = 0;
        while(i < ch.length && j < ch.length){
            if(ch[j] == ch[i]){
                res[j] = i+1;
                i++;
                j++;
            }else{
                if(i == 0){
                    res[j] = 0;
                    j++;
                }else{
                    i = res[i-1];
                }
            }
        }
        return res;
    }
```

written by chnsht original link here

Solution 3

Basic idea:

1. First char of input string is first char of repeated substring
2. Last char of input string is last char of repeated substring
3. Let S1 = S + S (where S in input string)
4. Remove 1 and last char of S1. Let this be S2
5. If S exists in S2 then return true else false
6. Let i be index in S2 where S starts then repeated substring length i + 1 and repeated substring S[0: i+1]

```
def repeatedSubstringPattern(self, str):

        """
        :type str: str
        :rtype: bool
        """
        if not str:
            return False

        ss = (str + str)[1:-1]
        return ss.find(str) != -1
```

written by rsrs3 original link here

From Leetcoder.

Minimum Moves to Equal Array Elements II

Given a **non-empty** integer array, find the minimum number of moves required to make all array elements equal, where a move is incrementing a selected element by 1 or decrementing a selected element by 1.

You may assume the array's length is at most 10,000.

Example:

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

Output:
2

Explanation:
Only two moves are needed (remember each move increments or decrements one element)
:

[1,2,3]  =>  [2,2,3]  =>  [2,2,2]
```

Solution 1

```
public class Solution {
    public int minMoves2(int[] nums) {
        Arrays.sort(nums);
        int i = 0, j = nums.length-1;
        int count = 0;
        while(i < j){
            count += nums[j]-nums[i];
            i++;
            j--;
        }
        return count;
    }
}
```

written by chnsht original link here

Solution 2

This solution relies on the fact that if we increment/decrement each element to the median of all the elements, the optimal number of moves is necessary. The median of all elements can be found in expected O(n) time using QuickSelect (or O(n) time using deterministic select).

'''

```
public int minMoves2(int[] nums) {
    int sum = 0;
    int median = findMedian(nums);
    for (int i=0;i<nums.length;i++) {
        sum += Math.abs(nums[i] - median);
    }
    return sum;
}

public int findMedian(int[] nums) {
    return getKth(nums.length/2+1, nums, 0, nums.length - 1);
}

public int getKth(int k, int[] nums, int start, int end) {
    int pivot = nums[end];
    int left = start;
    int right = end;

    while (true) {
        while (nums[left] < pivot && left < right) left++;
        while (nums[right] >= pivot && right > left) right--;
        if (left == right) break;
        swap(nums, left, right);
    }

    swap(nums, left, end);
    if (k == left + 1)  return pivot;
    else if (k < left + 1) return getKth(k, nums, start, left - 1);
    else return getKth(k, nums, left + 1, end);
}

public void swap(int[] nums, int n1, int n2) {
 int tmp = nums[n1];
 nums[n1] = nums[n2];
 nums[n2] = tmp;
}
```

'''

written by compton_scatter original link here

Solution 3

```
def minMoves2(self, nums):
    median = sorted(nums)[len(nums) / 2]
    return sum(abs(num - median) for num in nums)

def minMoves2(self, nums):
    nums.sort()
    return sum(nums[~i] - nums[i] for i in range(len(nums) / 2))
```

written by StefanPochmann original link here

From Leetcoder.

Island Perimeter

You are given a map in form of a two-dimensional integer grid where 1 represents land and 0 represents water. Grid cells are connected horizontally/vertically (not diagonally). The grid is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells). The island doesn't have "lakes" (water inside that isn't connected to the water around the island). One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

Example:

```
[[0,1,0,0],
 [1,1,1,0],
 [0,1,0,0],
 [1,1,0,0]]

Answer: 16
Explanation: The perimeter is the 16 yellow stripes in the image below:
```

Solution 1

1. loop over the matrix and count the number of islands;
2. if the current dot is an island, count if it has any right neighbour or down neighbour;
3. the result is islands * 4 - neighbours * 2

```
public class Solution {
    public int islandPerimeter(int[][] grid) {
        int islands = 0, neighbours = 0;

        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                if (grid[i][j] == 1) {
                    islands++; // count islands
                    if (i < grid.length - 1 && grid[i + 1][j] == 1) neighbours++;
// count down neighbours
                    if (j < grid[i].length - 1 && grid[i][j + 1] == 1) neighbours
++; // count right neighbours
                }
            }
        }

        return islands * 4 - neighbours * 2;
    }
}
```

written by liupangzi original link here

Solution 2

1. find how many 1 in the map. If without the consideration of surrounding cells, the total perimeter should be the total amount of 1 times 4.
2. find how many cell walls that connect with both lands. We need to deduct twice of those lines from total perimeter

```
int islandPerimeter(vector<vector<int>>& grid) {
        int count=0, repeat=0;
        for(int i=0;i<grid.size();i++)
        {
            for(int j=0; j<grid[i].size();j++)
                {
                    if(grid[i][j]==1)
                    {
                        count ++;
                        if(i!=0 && grid[i-1][j] == 1) repeat++;
                        if(j!=0 && grid[i][j-1] == 1) repeat++;
                    }
                }
        }
        return 4*count-repeat*2;
    }
```

@msg thanks for the edit

written by beckswu original link here

Solution 3

```
public static int islandPerimeter(int[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) return 0;
        int result = 0;
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[0].length; j++) {
                if (grid[i][j] == 1) {
                    result += 4;
                    if (i > 0 && grid[i-1][j] == 1) result -= 2;
                    if (j > 0 && grid[i][j-1] == 1) result -= 2;
                }
            }
        }
        return result;
    }
```

written by dreamchase original link here

From Leetcoder.

Can I Win

In the "100 game," two players take turns adding, to a running total, any integer from 1..10. The player who first causes the running total to reach or exceed 100 wins.

What if we change the game so that players cannot re-use integers?

For example, two players might take turns drawing from a common pool of numbers of 1..15 without replacement until they reach a total >= 100.

Given an integer `maxChoosableInteger` and another integer `desiredTotal`, determine if the first player to move can force a win, assuming both players play optimally.

You can always assume that `maxChoosableInteger` will not be larger than 20 and `desiredTotal` will not be larger than 300.

Example

```
Input:
maxChoosableInteger = 10
desiredTotal = 11

Output:
false

Explanation:
No matter which integer the first player choose, the first player will lose.
The first player can choose an integer from 1 up to 10.
If the first player choose 1, the second player can only choose integers from 2 up
to 10.
The second player will win by choosing 10 and get a total = 11, which is >= desired
Total.
Same with other integers chosen by the first player, the second player will always
win.
```

Solution 1

After solving several "Game Playing" questions in leetcode, I find them to be pretty similar. Most of them can be solved using the **top-down DP** approach, which "brute-forcely" simulates every possible state of the game.

The key part for the top-down dp strategy is that we need to**avoid repeatedly solving sub-problems**. Instead, we should use some strategy to "remember" the outcome of sub-problems. Then when we see them again, we instantly know their result. By doing this, ~~we can always reduce time complexity from **exponential** to **polynomial**~~.
(**EDIT:** Thanks for @billbirdh for pointing out the mistake here. For this problem, by applying the memo, we at most compute for every subproblem once, and there are `O(2^n)` subproblems, so the complexity is `O(2^n)` after memorization. (Without memo, time complexity should be like `O(n!)`)

For this question, the key part is: `what is the state of the game`? Intuitively, to uniquely determine the result of any state, we need to know:

1. The unchosen numbers
2. The remaining desiredTotal to reach

A second thought reveals that **1)** and **2)** are actually related because we can always get the **2)** by deducting the sum of chosen numbers from original desiredTotal.

Then the problem becomes how to describes the state using**1)**.

In my solution, I use a **boolean array** to denote which numbers have been chosen, and then a question comes to mind, if we want to use a Hashmap to remember the outcome of sub-problems, can we just use `Map<boolean[], Boolean>` ? **Obviously we cannot**, because the if we use boolean[] as a key, the reference to boolean[] won't reveal the actual content in boolean[].

Since in the problem statement, it says `maxChoosableInteger` will not be larger than `20`, which means the length of our **boolean[] array** will be less than `20`. Then we can use an `Integer` to represent this boolean[] array. How?

Say the boolean[] is `{false, false, true, true, false}`, then we can transfer it to an Integer with binary representation as `00110`. Since Integer is a perfect choice to be the key of HashMap, then we now can "memorize" the sub-problems using `Map<Integer, Boolean>`.

The rest part of the solution is just simulating the game process using the top-down dp.

```java
public class Solution {
    Map<Integer, Boolean> map;
    boolean[] used;
    public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
        int sum = (1+maxChoosableInteger)*maxChoosableInteger/2;
        if(sum < desiredTotal) return false;
        if(desiredTotal <= 0) return true;

        map = new HashMap();
        used = new boolean[maxChoosableInteger+1];
        return helper(desiredTotal);
    }

    public boolean helper(int desiredTotal){
        if(desiredTotal <= 0) return false;
        int key = format(used);
        if(!map.containsKey(key)){
    // try every unchosen number as next step
            for(int i=1; i<used.length; i++){
                if(!used[i]){
                    used[i] = true;
     // check whether this lead to a win (i.e. the other player lose)
                    if(!helper(desiredTotal-i)){
                        map.put(key, true);
                        used[i] = false;
                        return true;
                    }
                    used[i] = false;
                }
            }
            map.put(key, false);
        }
        return map.get(key);
    }

// transfer boolean[] to an Integer
    public int format(boolean[] used){
        int num = 0;
        for(boolean b: used){
            num <<= 1;
            if(b) num |= 1;
        }
        return num;
    }
}
```

Updated: Thanks for @ckcz123 for sharing the great idea. In Java, to denote `boolean[]`, an easier way is to use `Arrays.toString(boolean[])`, which will transfer a `boolean[]` to sth like `"[true, false, false, ....]"`, which is also not limited to how `maxChoosableInteger` is set, so it can be generalized to arbitrary large `maxChoosableInteger`.

written by leogogogo original link here

Solution 2

```java
public class Solution {

    private Boolean[] win;
    int choosen = 0;

    public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
        if (desiredTotal == 0) {
            return true;
        }
        if ((1 + maxChoosableInteger) * maxChoosableInteger / 2 < desiredTotal) {
            return false;
        }
        win = new Boolean[1 << maxChoosableInteger];
        return canWin(maxChoosableInteger, desiredTotal, 0);
    }

    private boolean canWin(int n, int total, int now) {
        if (win[choosen] != null)
            return win[choosen];
        if (now >= total) {
            win[choosen] = false;
            return false;
        }
        for (int i = 1; i <= n; i++) {
            int bit = 1 << (i - 1);
            if ((choosen & bit) == 0) {
                choosen ^= bit;
                boolean ulose = !canWin(n, total, now + i);
                choosen ^= bit;

                if (ulose) {
                    win[choosen] = true;
                    return true;
                }
            }
        }
        win[choosen] = false;
        return false;
    }
}
```

written by wzdxt original link here

Solution 3

The solution is quite strightforward. First off we have to eliminate primitive cases. So,

- if the first player can choose a number, which is already greater than or equal to the desired total obviously it wins.
- If max choosable integer is less than the desired total, but if it exceeds the desired total in sum with any other number then the first player looses anyway.
- If the sum of all number in the pool cannot exceed or reach the desired total, then no one can win.

Now, for the other cases we can use MiniMax logic to reveal the winner. Because both player play optimally, In order to win, the first player has to make a choice, that leaves the second player no chance to win. Thus, at each step we consider all the possible choices by the current player and give turn to the second player recursively. If we find a move, after which the second player looses anyway or we have already exceed the desired total by adding the chosen number, we return true, i.e the current player wins. This way the game looks like the following tree:

```
     player1 ->  0
               /| ...\
  player2 -> 1 2 ....max
            /|\ ..../ | \
player1 -> 2 3...  1  2 ..max-1
           ...                \
player1 ->   /      |     \   loose
player2 -> loose   win   loose
```

The figure above helps to imagine how the algorithm considers all possible scenarios of the game. The leafs of the game tree are loose or win states for one of the players. Finally the logic concludes to the idea, that if some branch does not contain any leaf that ends with win state for player2, the move associated with that branch is the optimal one for the first player.

P.S: Time complexity of naive implementation will work for O(n!). Therefore we have to memorize branch states after traversing once.

```
public class Solution {
    Map<Integer, Boolean> set[];
    public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
        if(maxChoosableInteger >= desiredTotal) return true;
        if(maxChoosableInteger+1 >=desiredTotal) return false;
        set = new Map[301];
        for(int i  =0 ;i<301;i++) set[i] = new HashMap<>();
        if(maxChoosableInteger*(maxChoosableInteger+1)/2 < desiredTotal) return f
alse;
        return canWin((1<<maxChoosableInteger+1)-1, desiredTotal);
    }

    public boolean canWin(int set1, int total){
        if(set[total].containsKey(set1)) return set[total].get(set1);
        for(int i = 20;i>=1;i--){
            int p = (1<<i);
            if((p&set1) == p){
                int set1next = (set1^p);
                int totalNext = total - i;
                if(totalNext<=0) return true;
                boolean x;
                if(set[totalNext].containsKey(set1next)) x = set[totalNext].get(s
et1next);
                else x = canWin(set1next, totalNext);
                if(!x){
                    set[total].put(set1, true);
                    return true;
                }
            }
        }
        set[total].put(set1, false);
        return false;
    }
}
```

written by ZhassanB original link here

From Leetcoder.

Optimal Account Balancing

A group of friends went on holiday and sometimes lent each other money. For example, Alice paid for Bill's lunch for $10. Then later Chris gave Alice $5 for a taxi ride. We can model each transaction as a tuple (x, y, z) which means person x gave person y $z. Assuming Alice, Bill, and Chris are person 0, 1, and 2 respectively (0, 1, 2 are the person's ID), the transactions can be represented as `[[0, 1, 10], [2, 0, 5]]`.

Given a list of transactions between a group of people, return the minimum number of transactions required to settle the debt.

Note:

1. A transaction will be given as a tuple (x, y, z). Note that `x ≠ y` and `z > 0`.
2. Person's IDs may not be linear, e.g. we could have the persons 0, 1, 2 or we could also have the persons 0, 2, 6.

Example 1:

```
Input:
[[0,1,10], [2,0,5]]

Output:
2

Explanation:
Person #0 gave person #1 $10.
Person #2 gave person #0 $5.

Two transactions are needed. One way to settle the debt is person #1 pays person #0
 and #2 $5 each.
```

Example 2:

```
Input:
[[0,1,10], [1,0,1], [1,2,5], [2,0,5]]

Output:
1

Explanation:
Person #0 gave person #1 $10.
Person #1 gave person #0 $1.
Person #1 gave person #2 $5.
Person #2 gave person #0 $5.

Therefore, person #1 only need to give person #0 $4, and all debt is settled.
```

Solution 1

Given the input [[0,1,1], [2,3,2], [4,5,3], [6,7,4], [8,9,5], [10,11,6], [12,13,7], [14,15,2], [14,16,2], [14,17,2], [14,18,2]]

The provided expected result is 14. However, there are only 11 transactions, so the upper bound of the result should be 11.

written by tcui original link here

Solution 2

Think about this case:
[[0,3,2],[1,4,3],[2,3,2],[2,4,2]]

The correct answer should be 3, but the so-called standard answer is 4.
Actually this question seems to be a NPC problem. And I don't know why there is no data scale.

written by nyunyunyunyu original link here

Solution 3

```
import java.util.*;

public class Solution {
    public int minTransfers(int[][] trans) {
        Map<Integer, Integer> net = new HashMap<>();
        for(int i = 0; i < trans.length; i++){
            net.put(trans[i][0], net.getOrDefault(trans[i][0], 0) - trans[i][2]);
            net.put(trans[i][1], net.getOrDefault(trans[i][1], 0) + trans[i][2]);
        }
        int[] temp = new int[net.size()];
        int i = 0;
        for(int j : net.values()){
            if(j != 0)temp[i++] = j;
        }
        int[] a = new int[i];
        System.arraycopy(temp, 0, a, 0, i);
        transactions.clear();
        number = Integer.MAX_VALUE;
        mintran(a, 0);
        return number;
    }

    private List<int[]> transactions = new ArrayList<>();
    private int number = Integer.MAX_VALUE;

    private void mintran(int[] a, int start){
        //System.out.println(Arrays.toString(a));
        if(transactions.size() >= number) return;
        if(number == (a.length + 1)/2) return;

        if(a.length < 2){
            number = 0;
            return;
        }else if(a.length == 2) {
            number = a[0] == 0 ? 0 : 1;
            return;
        }else{
            int ind = -1;
            int max = Integer.MIN_VALUE;
            int i = start;
            for(; i < a.length; i++){
                if(Math.abs(a[i]) > max){
                    max = Math.abs(a[i]);
                    ind = i;
                }
            }

            if(max == 0 || start == a.length){
                if(transactions.size() < number){
                    number = transactions.size();
                }
                return;
            }
```

```
            int temp = a[ind];
            a[ind] = a[start];
            a[start] = temp;

            for(i = start + 1; i < a.length; i++){
                if(a[i] * a[start] < 0) {
                    transactions.add(new int[]{a[i], a[start]});
                    temp = a[i];
                    a[i] += a[start];
                    mintran(a, start + 1);
                    a[i] = temp;
                    transactions.remove(transactions.size()-1);
                }
            }

            temp = a[ind];
            a[ind] = a[start];
            a[start] = temp;

        }
    }

    public static void main(String[] args) {
        int[][] A = new int[][] {{0,1,1}, {2,3,2}, {4,5,3}, {6,7,4}, {8,9,5}, {1
0,11,6}, {12,13,7}, {14,15,2}, {14,16,2}, {14,17,2}, {14,18,2}};
        Solution solution = new Solution();
        System.out.println(solution.minTransfers(A));
    }
}
```

written by fatalme original link here

Count The Repetitions

Define `S = [s,n]` as the string S which consists of n connected strings s. For example, `["abc", 3]` ="abcabcabc".

On the other hand, we define that string s1 can be obtained from string s2 if we can remove some characters from s2 such that it becomes s1. For example, “abc” can be obtained from “abdbec” based on our definition, but it can not be obtained from “acbbe”.

You are given two non-empty strings s1 and s2 (each at most 100 characters long) and two integers $0 \le n1 \le 10^6$ and $1 \le n2 \le 10^6$. Now consider the strings S1 and S2, where `S1=[s1,n1]` and `S2=[s2,n2]`. Find the maximum integer M such that `[S2,M]` can be obtained from `S1`.

Example:

```
Input:
s1="acb", n1=4
s2="ab", n2=2

Return:
2
```

Solution 1

I didn't come up with any good solution so I tried brute force. Key points:

1. How do we know "string s2 can be obtained from string s1"? Easy, use two pointers iterate through s2 and s1. If chars are equal, move both. Otherwise only move pointer1.
2. We repeat step 1 and go through s1 for n1 times and count how many times can we go through s2.
3. Answer to this problem is times go through s2 divide by n2.

```
public class Solution {
    public int getMaxRepetitions(String s1, int n1, String s2, int n2) {
        char[] array1 = s1.toCharArray(), array2 = s2.toCharArray();
        int count1 = 0, count2 = 0, i = 0, j = 0;

        while (count1 < n1) {
            if (array1[i] == array2[j]) {
                j++;
                if (j == array2.length) {
                    j = 0;
                    count2++;
                }
            }
            i++;
            if (i == array1.length) {
                i = 0;
                count1++;
            }
        }

        return count2 / n2;
    }
}
```

written by shawngao original link here

Solution 2

IDEA:

Given a str2, for each str, we can give a value v to this str such that, after greedily looking through str, our imaginary next step is to find str2[v].
In our problem, str is always (str1,n), with a given str1, so, we can take one more step and say that for each n, there is a unique v associated to n(i.e to (str,n)).

define a division and a modulo between two strings as follow:

str/str2=argmax{i, (str2,i) can be obtained by str}
str%str2=the v mentioned above associated with str.

All possible values of v is less than str2.size(),
so (str1,n)%str2 will begin to **repeat a pattern** after a certain n less than str2.size().
(the pattern is the same because in the cases with the same v, our situations are exactly the same),
so is (str1,n)/str2-(str1,n+1)/str2 for the same reason.
We can therefore precompute a table for all these values with O(str1.length*str2.length).

(str1,n) can be divided in three parts:

sth before pattern(A) + pattern parts(B) + sth after pattern(C)

The pattern does not necessarily begin in the first str1, we shall see if n is great enough so that there can be a pattern.

The last pattern(C) is not necessarily complete, we need to calculate it separately.

We can finish in just looking to the precomputed table and doing some simple maths.

```
class Solution {
public:
    int getMaxRepetitions(string s1, int n1, string s2, int n2) {
        vector<int> rapport(102,-1);
        vector<int> rest(102,-1);
        int b=-1;int posRest=0;int rap=0;
        int last=-1;
        rapport[0]=rest[0]=0;//case when n=0
        for(int i=1;i<=s2.size()+1;i++){
            int j;
            for(j=0;j<s1.size();j++){
                if(s2[posRest]==s1[j]){
                    posRest++;
                    if(posRest==s2.size()){
                        rap++;
                        posRest=0;
                    }
                }
            }
            for(int k=0;k<i;k++){
                if(posRest==rest[k]){b=k;last=i;break;}
            }
            rapport[i]=rap;rest[i]=posRest;
            if(b>=0)break;
        }
        int interval=last-b;
        if(b>=n1)return rapport[n1]/n2;
        return ((n1-b)/interval*(rapport[last]-rapport[b])+rapport[(n1-b)%interva
l+b])/n2;
//corrected thanks to @zhiqing_xiao and @iaming
    }
};
```

written by 70664914 original link here

Solution 3

The key is, we just need to calculate ***what will remain after s1 obtains s2*** .

That is, `(s1, s2) -> (sRemain, matchCnt)` ; for example,
`(abcd, ab) -> (cd, 1)`
`(ababa, ab) -> (a, 2)`
`(a, aaaa) -> (a, 0)`
`(aabaabaab, aba) -> (ab, 2)` as `aabaaba` exactly matches `aba` twice.

And, each time we append `s1` to the `remain string` , to make a sequence: (Using `[]` to mark the `remain string`)
`(abcd, ab): abcd -> [cd]abcd -> [cd]abcd -> [cd]abcd -> ...`
`(ababa, ab): ababa -> [a]ababa -> [a]ababa -> [a]ababa -> ...`
`(a, aaaa): a -> [a]a -> [aa]a -> [aaa]a -> a -> [a]a -> [aa]a -> ...`
`(aabaabaab, aba): aabaabaab -> [ab]aabaabaab -> [ab]aabaabaab -> ...`

Obviously, there will be a loop in the sequence, assume the length of loop is `loop` , and the length before the loop is `k` .
`(abcd, ab): loop=1, k=1`
`(a, aaaa): loop=4, k=0`
`(aabaabaab, aba): loop=1, k=1`

So, we just need to calculate `the count of each loop` , and `the count before entering the loop` , and calculate the total.

Here is the code with detailed comment, 21ms.

```
public class Solution {
    public int getMaxRepetitions(String s1, int n1, String s2, int n2) {
        if (!ableToObtain(s1, s2)) return 0; // check if [s1. ∞] obtains s2
        int cnt=0, k=-1;
        String s=s1;
        StringBuilder remainBuilder; // record `remain string`
        ArrayList<String> stringList=new ArrayList<>(); // record all the `remain
string`
        ArrayList<Integer> countList=new ArrayList<>(); // record matching count
from start to the current remain string
        stringList.add(""); // record empty string
        countList.add(0);
        for (int i=0;i<=n1;i++) {
            remainBuilder=new StringBuilder();
            cnt+=getRemain(s, s2, remainBuilder); // get the next remain string,
returns the count of matching
            String remain=remainBuilder.toString();
            if ((k=stringList.indexOf(remain))!=-1) break; // if there is a loop,
break
            stringList.add(remain); // record the remain string into arraylist
            countList.add(cnt);
            s=remain+s1; // append s1 to make a new string
        }
        // here, k is the beginning of the loop
        if (k==-1) return cnt/n2; // if there is no loop
        int countOfLoop=cnt-countList.get(k), loopLength=stringList.size()-k; //
```

```
        int countOfLoop=cnt-countList.get(k), loopLength=stringList.size()-k; //
get matching count in the loop, and loop length
        cnt=countList.get(k);
        n1-=k;
        cnt+=countOfLoop*(n1/loopLength);
        n1%=loopLength;
        cnt+=countList.get(k+n1)-countList.get(k);
        return cnt/n2;
    }

    // check if [s1. ∞] obtains s2
    private boolean ableToObtain(String s1, String s2) {
        boolean[] cnt=new boolean[26];
        for (char c: s1.toCharArray()) cnt[c-'a']=true;
        for (char c: s2.toCharArray()) {
            if (!cnt[c-'a']) return false;
        }
        return true;
    }

    // get remain string after s1 obtains s2, return the matching count
    private int getRemain(String s1, String s2, StringBuilder remain) {
        int cnt=0, lastMatch=-1, p2=0;
        for (int p1=0;p1<s1.length();p1++) {
            if (s1.charAt(p1)==s2.charAt(p2)) {
                if (++p2==s2.length()) {
                    p2=0;
                    cnt++;
                    lastMatch=p1;
                }
            }
        }
        remain.append(s1.substring(lastMatch+1));
        return cnt;
    }
}
```

written by ckcz123 original link here

From Leetcoder.

Unique Substrings in Wraparound String

Consider the string `s` to be the infinite wraparound string of "abcdefghijklmnopqrstuvwxyz", so `s` will look like this: "...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".

Now we have another string `p`. Your job is to find out how many unique non-empty substrings of `p` are present in `s`. In particular, your input is the string `p` and you need to output the number of different non-empty substrings of `p` in the string `s`.

Note: `p` consists of only lowercase English letters and the size of p might be over 10000.

Example 1:

```
Input: "a"
Output: 1

Explanation: Only the substring "a" of string "a" is in the string s.
```

Example 2:

```
Input: "cac"
Output: 2
Explanation: There are two substrings "a", "c" of string "cac" in the string s.
```

Example 3:

```
Input: "zab"
Output: 6
Explanation: There are six substrings "z", "a", "b", "za", "ab", "zab" of string "z
ab" in the string s.
```

Solution 1

After failed with pure math solution and time out with DFS solution, I finally realized that this is a DP problem...
The idea is, if we know the max number of unique substrings in `p` ends with `'a', 'b', ..., 'z'`, then the summary of them is the answer. Why is that?

1. The max number of unique substring ends with a letter equals to the length of max contiguous substring ends with that letter. Example `"abcd"`, the max number of unique substring ends with `'d'` is 4, apparently they are `"abcd", "bcd", "cd" and "d"`.
2. If there are overlapping, we only need to consider the longest one because it covers all the possible substrings. Example: `"abcdbcd"`, the max number of unique substring ends with `'d'` is 4 and all substrings formed by the 2nd `"bcd"` part are covered in the 4 substrings already.
3. No matter how long is a contiguous substring in `p`, it is in `s` since `s` has infinite length.
4. Now we know the max number of unique substrings in `p` ends with `'a', 'b', ..., 'z'` and those substrings are all in `s`. Summary is the answer, according to the question.

Hope I made myself clear...

```java
public class Solution {
    public int findSubstringInWraproundString(String p) {
        // count[i] is the maximum unique substring end with ith letter.
        // 0 - 'a', 1 - 'b', ..., 25 - 'z'.
        int[] count = new int[26];

        // store longest contiguous substring ends at current position.
        int maxLengthCur = 0;

        for (int i = 0; i < p.length(); i++) {
            if (i > 0 && (p.charAt(i) - p.charAt(i - 1) == 1 || (p.charAt(i - 1)
- p.charAt(i) == 25))) {
                maxLengthCur++;
            }
            else {
                maxLengthCur = 1;
            }

            int index = p.charAt(i) - 'a';
            count[index] = Math.max(count[index], maxLengthCur);
        }

        // Sum to get result
        int sum = 0;
        for (int i = 0; i < 26; i++) {
            sum += count[i];
        }
        return sum;
    }
}
```

written by shawngao original link here

Solution 2

```
int findSubstringInWraproundString(string p) {
        vector<int> letters(26, 0);
        int res = 0, len = 0;
        for (int i = 0; i < p.size(); i++) {
            int cur = p[i] - 'a';
            if (i > 0 && p[i - 1] != (cur + 26 - 1) % 26 + 'a') len = 0;
            if (++len > letters[cur]) {
                res += len - letters[cur];
                letters[cur] = len;
            }
        }
        return res;
    }
```

written by zyoppy008 original link here

Solution 3

Hi there! I am sharing two different solutions with explanations.

Well, the first solution idea is Optimized brute force. The 'naivest' way is for each substring of p in alphabetic order (with rotation %26) brute force all substrings and add to some set. Then at last just return size of the set. That method works but it has two problems, they are:

- Memory limit
- Time limit

Because it works for O(n^3) time and O(n^2) space.
How can we optimize memory? Because we are considering only strings in alphabetic order, it is sufficient to remember the first character index, last character index and the length for each found substring. This way we replace set of strings to 3D boolean array.
Well, how can we optimize time complexity? We can increase performance of counting unique substrings for already found substring, which size is greater than 26. The evidence mentioned above helps us again, because we just need to consider combinations of first 26 character and the last 26 characters for different length, by incrementing length by 26. Such a way we get algorithm that runs for O(26*n) time and O((26^2)*n) space complexities.

```
public class Solution {
    public int findSubstringInWraproundString(String p) {
        if(p == null || p.isEmpty()) return 0;
        boolean [][][] set = new boolean[26][26][p.length()+1];
        int i = 0;
        int count = 0;
        boolean [] visited = new boolean[26];
        StringBuilder build = new StringBuilder();
        char [] s = p.toCharArray();
        int n = s.length;
        while(i<n){
            char prev = s[i];
            build.append(prev);
            i++;
            while(i<p.length() && s[i]-'a' == (prev-'a'+1)%26){
                prev = s[i];
                build.append(prev);
                i++;
            }

            String next = build.toString();
            int l = next.charAt(0)-'a';
            int r = next.charAt(next.length()-1)-'a';
            if(!set[l][r][next.length()]){
                count++;
                set[l][r][next.length()] = true;
                count+= countUniqueSubstr(next, set);
            }
            build.setLength(0);
        }
```

```
        return count;
    }


    public int countUniqueSubstr(String str, boolean [][][] set){
        int count = 0;
        if(str.length()>26){
            int n = str.length();
            for(int i = 0;i<26;i++){
                int l = str.charAt(i)-'a';
                for(int j = n-1;j>=n-26;j--){
                    int r = str.charAt(j)-'a';
                    int limit = j-i+1;
                    int start = r-l+1;
                    if(r<l){
                        start = 27-l+r;
                    }
                    for(int size = start;size<=limit;size+=26){
                        if(!set[l][r][size]){
                            count++;
                            set[l][r][size] = true;
                        }
                    }
                }
            }

        } else {
            for(int size = 1;size<str.length();size++){
                for(int i = 0;i<=str.length()-size;i++){
                    String s = str.substring(i, i+size);
                    int l = s.charAt(0)-'a';
                    int r = s.charAt(s.length()-1)-'a';
                    if(!set[l][r][s.length()]){
                        count++;
                        set[l][r][s.length()] = true;
                    }
                }
            }
        }
        return count;
    }
}
```

The second solution is simple, just keep track of maximum length of alphabetic substrings ending at certain character then sum them up.

```java
public class Solution {
    public int findSubstringInWraproundString(String p) {
        if(p == null || p.isEmpty()) return 0;
        int dp[] = new int[26];
        int i = 0;
        int n = p.length();
        char [] s = p.toCharArray();
        int len = 1;
        while(i<n){
            char prev = s[i];
            i++;
            dp[prev - 'a'] = Math.max(dp[prev-'a'], len);
            while(i<p.length() && s[i]-'a' == (prev-'a'+1)%26){
                prev = s[i];
                len++;
                i++;
                dp[prev - 'a'] = Math.max(dp[prev-'a'], len);
            }
            dp[prev - 'a'] = Math.max(dp[prev-'a'], len);
            len = 1;
        }
        int count = 0;
        for(int j = 0;j<26;j++) count+=dp[j];
        return count;
    }
}
```

P.S: Sorry for poor and dirty code. I hope it will be understandable anyway

written by ZhassanB original link here

From Leetcoder.

Validate IP Address

In this problem, your job to write a function to check whether a input string is a valid IPv4 address or IPv6 address or neither.

IPv4 addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots (."), e.g., `172.16.254.1`;

Besides, you need to keep in mind that leading zeros in the IPv4 is illegal. For example, the address `172.16.254.01` is illegal.

IPv6 addresses are represented as eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons (":"). For example, the address `2001:0db8:85a3:0000:0000:8a2e:0370:7334` is a legal one. Also, we could omit some leading zeros among four hexadecimal digits and some low-case characters in the address to upper-case ones, so `2001:db8:85a3:0:0:8A2E:0370:7334` is also a valid IPv6 address(Omit leading zeros and using upper cases).

However, we don't replace a consecutive group of zero value with a single empty group using two consecutive colons (::) to pursue simplicity. For example, `2001:0db8:85a3::8A2E:0370:7334` is an invalid IPv6 address.

Besides, you need to keep in mind that extra leading zeros in the IPv6 is also illegal. For example, the address `02001:0db8:85a3:0000:0000:8a2e:0370:7334` is also illegal.

Note: You could assume there is no extra space in the test cases and there may some special characters in the input string.

Example 1:

```
Input: "172.16.254.1"

Output: "IPv4"

Explanation: This is a valid IPv4 address, return "IPv4".
```

Example 2:

```
Input: "2001:0db8:85a3:0:0:8A2E:0370:7334"

Output: "IPv6"

Explanation: This is a valid IPv6 address, return "IPv6".
```

Example 3:

Input: "256.256.256.256"

Output: "Neither"

Explanation: This is neither a IPv4 address nor a IPv6 address.

Solution 1

```
public String validIPAddress(String IP) {
 if(isValidIPv4(IP)) return "IPv4";
 else if(isValidIPv6(IP)) return "IPv6";
 else return "Neither";
}

public boolean isValidIPv4(String ip) {
 if(ip.length()<7) return false;
 if(ip.charAt(0)=='.') return false;
 if(ip.charAt(ip.length()-1)=='.') return false;
 String[] tokens = ip.split("\\.");
 if(tokens.length!=4) return false;
 for(String token:tokens) {
  if(!isValidIPv4Token(token)) return false;
 }
 return true;
}
public boolean isValidIPv4Token(String token) {
 if(token.startsWith("0") && token.length()>1) return false;
 try {
  int parsedInt = Integer.parseInt(token);
  if(parsedInt<0 || parsedInt>255) return false;
  if(parsedInt==0 && token.charAt(0)!='0') return false;
 } catch(NumberFormatException nfe) {
  return false;
 }
 return true;
}

public boolean isValidIPv6(String ip) {
 if(ip.length()<15) return false;
 if(ip.charAt(0)==':') return false;
 if(ip.charAt(ip.length()-1)==':') return false;
 String[] tokens = ip.split(":");
 if(tokens.length!=8) return false;
 for(String token: tokens) {
  if(!isValidIPv6Token(token)) return false;
 }
 return true;
}
public boolean isValidIPv6Token(String token) {
 if(token.length()>4 || token.length()==0) return false;
 char[] chars = token.toCharArray();
 for(char c:chars) {
  boolean isDigit = c>=48 && c<=57;
  boolean isUppercaseAF = c>=65 && c<=70;
  boolean isLowerCaseAF = c>=97 && c<=102;
  if(!(isDigit || isUppercaseAF || isLowerCaseAF))
   return false;
 }
 return true;
}
```

written by fabrizio3 original link here

Solution 2

```
const ip4 = /^([1-9]\d{0,2}|0)(?:\.([1-9]\d{0,2}|0)){3}$/;
const ip6 = /^([0-9a-fA-F]{1,4})(\:[0-9a-fA-F]{1,4}){7}$/;

var validIPAddress = function(IP) {
    const isIp4 = ip4.exec(IP);
    if (isIp4 && isIp4.slice(1).every(d => parseInt(d, 10) < 256))
        return 'IPv4';

    const isIp6 = ip6.exec(IP);
    if (isIp6)
        return 'IPv6';

    return 'Neither';
};
```

written by dettier original link here

Solution 3

```
class Solution(object):
    def validIPAddress(self, IP):
        def is_hex(s):
            hex_digits = set("0123456789abcdefABCDEF")
            for char in s:
                if not (char in hex_digits):
                    return False
            return True
        ary = IP.split('.')
        if len(ary) == 4:
            for i in xrange(len(ary)):
                if not ary[i].isdigit() or not 0 <= int(ary[i]) < 256 or (ary[i][
0] == '0' and len(ary[i]) > 1):
                    return "Neither"
            return "IPv4"
        ary = IP.split(':')
        if len(ary) == 8:
            for i in xrange(len(ary)):
                tmp = ary[i]
                if len(tmp) == 0 or not len(tmp) <= 4 or not is_hex(tmp):
                    return "Neither"
            return "IPv6"
        return "Neither"
```

written by Ipeq1 original link here

From Leetcoder.

Convex Polygon

Given a list of points that form a polygon when joined sequentially, find if this polygon is convex (Convex polygon definition).

Note:

1. There are at least 3 and at most 10,000 points.
2. Coordinates are in the range -10,000 to 10,000.
3. You may assume the polygon formed by given points is always a simple polygon (Simple polygon definition). In other words, we ensure that exactly two edges intersect at each vertex, and that edges otherwise **don't intersect each other**.

Example 1:

```
[[0,0],[0,1],[1,1],[1,0]]

Answer: True

Explanation:
```

Example 2:

```
[[0,0],[0,10],[10,10],[10,0],[5,5]]

Answer: False

Explanation:
```

Solution 1

get the **cross product** of the sequential input edge a, b as **tmp**, then:

if **tmp** == 0, a -> b 180° on the same line;
elif **tmp** > 0, a -> b clockwise;
else **tmp** < 0, a -> anticlockwise;

tmp = (p1[0]-p0[0])*(p2[1]-p0[1])-(p2[0]-p0[0])*(p1[1]-p0[1])*

```
class Solution(object):
    def isConvex(self, points):
        last = 0
        for i in xrange(2, len(points) + 2):
            p0, p1, p2 = points[(i - 2) % len(points)], points[(i - 1) % len(poin
ts)], points[i % len(points)]
            tmp = (p1[0]-p0[0])*(p2[1]-p0[1])-(p2[0]-p0[0])*(p1[1]-p0[1])
            if last * tmp < 0:
                return False
            last = tmp
        return last * tmp >= 0
```

Update instead of just maintaining the sequential cross product result, any of the two cross product shouldn't multiplies to minus:

```
class Solution(object):
    def isConvex(self, points):
        last, tmp = 0, 0
        for i in xrange(2, len(points) + 3):
            p0, p1, p2 = points[(i - 2) % len(points)], points[(i - 1) % len(poin
ts)], points[i % len(points)]
            tmp = (p1[0]-p0[0])*(p2[1]-p0[1])-(p2[0]-p0[0])*(p1[1]-p0[1])
            if tmp:
                if last * tmp < 0:
                    return False
                last = tmp
        return True
```

written by Ipeq1 original link here

Solution 2

Well, I have to say that this problem is beyond my knowledge.@Ipeq1 and @zzg_zzm have explained how to solve this problem in their posts
https://discuss.leetcode.com/topic/70643/i-believe-this-time-it-s-far-beyond-my-ability-to-get-a-good-grade-of-the-contest
https://discuss.leetcode.com/topic/70664/c-7-line-o-n-solution-to-check-convexity-with-cross-product-of-adajcent-vectors-detailed-explanation
The algorithm itself is not hard but I have no idea there exists such a way to determine if a polygon is convex or not. Laugh at me for my ignorance... I believe 90% of programmers can solve this problem if they were given the formula.
Anyway, following is the Java solution with in-line explanation. Accepted, 32ms.
Reference: http://csharphelper.com/blog/2014/07/determine-whether-a-polygon-is-convex-in-c/

```java
public class Solution {
    public boolean isConvex(List<List<Integer>> points) {
        // For each set of three adjacent points A, B, C, find the cross product
AB · BC. If the sign of
        // all the cross products is the same, the angles are all positive or nega
tive (depending on the
        // order in which we visit them) so the polygon is convex.
        boolean gotNegative = false;
        boolean gotPositive = false;
        int numPoints = points.size();
        int B, C;
        for (int A = 0; A < numPoints; A++) {
            // Trick to calc the last 3 points: n - 1, 0 and 1.
            B = (A + 1) % numPoints;
            C = (B + 1) % numPoints;

            int crossProduct =
                crossProductLength(
                    points.get(A).get(0), points.get(A).get(1),
                    points.get(B).get(0), points.get(B).get(1),
                    points.get(C).get(0), points.get(C).get(1));
            if (crossProduct < 0) {
                gotNegative = true;
            }
            else if (crossProduct > 0) {
                gotPositive = true;
            }
            if (gotNegative && gotPositive) return false;
        }

        // If we got this far, the polygon is convex.
        return true;
    }

    // Return the cross product AB x BC.
    // The cross product is a vector perpendicular to AB and BC having length |AB
| * |BC| * Sin(theta) and
    // with direction given by the right-hand rule. For two vectors in the X-Y pla
ne, the result is a
    // vector with X and Y components 0 so the Z component gives the vector's len
gth and direction.
    private int crossProductLength(int Ax, int Ay, int Bx, int By, int Cx, int Cy)
    {
        // Get the vectors' coordinates.
        int BAx = Ax - Bx;
        int BAy = Ay - By;
        int BCx = Cx - Bx;
        int BCy = Cy - By;

        // Calculate the Z coordinate of the cross product.
        return (BAx * BCy - BAy * BCx);
    }
}
```

written by shawngao original link here

Solution 3

Re: I believe this time it's far beyond my ability to get a good grade of the contest!

Great solution inspired by @Ipeq1! Here is a C++ version with extracted determinant calculation.

The key observation for convexity is that **vector p_{i+1}-p_i always turns to the same direction to p_{i+2}-p_i formed by any 3 sequentially adjacent vertices, i.e., cross product (p_{i+1}-p_i) x (p_{i+2}-p_i) does not change sign when traversing sequentially along polygon vertices**.

Note that for any 2D vectors $\mathbf{v_1}$, $\mathbf{v_2}$,

- $\mathbf{v_1 \times v_2 = det([v_1, v_2])}$

which is the determinant of 2x2 matrix $\mathbf{[v_1, v_2]}$. And the sign of $\mathbf{det([v_1, v_2])}$ represents the positive z-direction of right-hand system from $\mathbf{v_1}$ to $\mathbf{v_2}$. So $\mathbf{det([v_1, v_2]) \geq 0}$ if and only if $\mathbf{v_1}$ turns at most 180 degrees **counterclockwise** to $\mathbf{v_2}$.

```
    bool isConvex(vector<vector<int>>& p) {
      long n = p.size(), prev = 0, cur;
      for (int i = 0; i < n; ++i) {
        vector<vector<int>> A; // = {p[(i+1)%n]-p[i], p[(i+2)%n]-p[i]}
        for (int j = 1; j < 3; ++j) A.push_back({p[(i+j)%n][0]-p[i][0], p[(i+j)%n
][1]-p[i][1]});
        if (cur = det2(A)) if (cur*prev < 0) return false; else prev = cur;
      }
      return true;
    }
    // calculate determinant of 2*2 matrix A
    long det2(vector<vector<int>>& A) { return A[0][0]*A[1][1] - A[0][1]*A[1][0];
}
```

written by zzg_zzm original link here

From Leetcoder.

Encode String with Shortest Length

Given a **non-empty** string, encode the string such that its encoded length is the shortest.

The encoding rule is: `k[encoded_string]`, where the *encoded_string* inside the square brackets is being repeated exactly *k* times.

Note:

1. *k* will be a positive integer and encoded string will not be empty or have extra space.
2. You may assume that the input string contains only lowercase English letters. The string's length is at most 160.
3. If an encoding process does not make the string shorter, then do not encode it. If there are several solutions, return any of them is fine.

Example 1:

```
Input: "aaa"
Output: "aaa"
Explanation: There is no way to encode it such that it is shorter than the input st
ring, so we do not encode it.
```

Example 2:

```
Input: "aaaaa"
Output: "5[a]"
Explanation: "5[a]" is shorter than "aaaaa" by 1 character.
```

Example 3:

```
Input: "aaaaaaaaaa"
Output: "10[a]"
Explanation: "a9[a]" or "9[a]a" are also valid solutions, both of them have the sam
e length = 5, which is the same as "10[a]".
```

Example 4:

```
Input: "aabcaabcd"
Output: "2[aabc]d"
Explanation: "aabc" occurs twice, so one answer can be "2[aabc]d".
```

Example 5:

```
Input: "abbbabbbcabbbabbbc"
Output: "2[2[abbb]c]"
Explanation: "abbbabbbc" occurs twice, but "abbbabbbc" can also be encoded to "2[ab
bb]c", so one answer can be "2[2[abbb]c]".
```

Solution 1

This is the first question I have answered in Leetcode. I hope you guys will like my solution. The approach here is simple. We will form 2-D array of Strings.
dp[i][j] = string from index i to index j in encoded form.

We can write the following formula as:-
dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j]) or if we can find some pattern in string from i to j which will result in more less length.

Time Complexity = O(n^3)

```
public class Solution {
```

```
 public String encode(String s) {
    String[][] dp = new String[s.length()][s.length()];

    for(int l=0;l<s.length();l++) {
        for(int i=0;i<s.length()-l;i++) {
            int j = i+l;
            String substr = s.substring(i, j+1);
            // Checking if string length < 5. In that case, we know that encoding
will not help.
            if(j - i < 4) {
                dp[i][j] = substr;
            } else {
                dp[i][j] = substr;
                // Loop for trying all results that we get after dividing the str
ings into 2 and combine the   results of 2 substrings
                for(int k = i; k<j;k++) {
                    if((dp[i][k] + dp[k+1][j]).length() < dp[i][j].length()){
                        dp[i][j] = dp[i][k] + dp[k+1][j];
                    }
                }

                // Loop for checking if string can itself found some pattern in i
t which could be repeated.
                for(int k=0;k<substr.length();k++) {
                    String repeatStr = substr.substring(0, k+1);
                    if(repeatStr != null
                       && substr.length()%repeatStr.length() == 0
                       && substr.replaceAll(repeatStr, "").length() == 0) {
                          String ss = substr.length()/repeatStr.length() + "[" +
dp[i][i+k] + "]";
                          if(ss.length() < dp[i][j].length()) {
                            dp[i][j] = ss;
                          }
                     }
                }
            }
        }
    }

    return dp[0][s.length()-1];
}
}
```

written by hatella original link here

Solution 2

3 for loop, so complexity O(n^3), bottom up DP with step goes from 1 to n, and for each step calculate all start and end locations. Use the collapse idea from another solution.

```
class Solution {
private:
 vector<vector<string>> dp;
public:
 string collapse(string& s, int i, int j) {
     string temp = s.substr(i, j - i + 1);
  auto pos = (temp+temp).find(temp, 1);
  if (pos >= temp.size()) {
      return temp;
  }
  return to_string(temp.size()/pos) + '['+ dp[i][i+pos-1]+']';
 }

 string encode(string s) {
  int n = s.size();
  dp = vector<vector<string>>(n, vector<string>(n, ""));
  for (int step = 1; step <= n; step++) {
   for (int i = 0; i + step - 1 < n; i++) {
    int j = i + step - 1;
    dp[i][j] = s.substr(i, step);
    for (int k = i; k < j; k++) {
     auto left = dp[i][k];
     auto right = dp[k + 1][j];
     if (left.size() + right.size() < dp[i][j].size()) {
      dp[i][j] = left + right;
     }
    }
    string replace = collapse(s, i, j);
    if (replace.size() < dp[i][j].size()) {
     dp[i][j] = replace;
    }
   }
  }
  return dp[0][n - 1];
 }
};
```

written by yanzhan2 original link here

Solution 3

Either don't encode `s` at all, or encode it as **one** part `k[...]` or encode it as **multiple** parts (in which case we can somewhere split it into two subproblems). Whatever is shortest. Uses @rsrs3's nice trick of searching `s` in `s + s`.

```
def encode(self, s, memo={}):
    if s not in memo:
        n = len(s)
        i = (s + s).find(s, 1)
        one = '%d[%s]' % (n / i, self.encode(s[:i])) if i < n else s
        multi = [self.encode(s[:i]) + self.encode(s[i:]) for i in xrange(1, n)]
        memo[s] = min([s, one] + multi, key=len)
    return memo[s]
```

written by StefanPochmann original link here

From Leetcoder.

Concatenated Words

Given a list of words (**without duplicates**), please write a program that returns all concatenated words in the given list of words.

A concatenated word is defined as a string that is comprised entirely of at least two shorter words in the given array.

Example:

```
Input: ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratca
tdogcat"]

Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]

Explanation: "catsdogcats" can be concatenated by "cats", "dog" and "cats";
 "dogcatsdog" can be concatenated by "dog", "cats" and "dog";
"ratcatdogcat" can be concatenated by "rat", "cat", "dog" and "cat".
```

Note:

1. The number of elements of the given array will not exceed `10,000`
2. The length sum of elements in the given array will not exceed `600,000`.
3. All the input string will only include lower case letters.
4. The returned elements order does not matter.

Solution 1

Do you still remember how did you solve this problem?
https://leetcode.com/problems/word-break/

If you do know one optimized solution for above question is using `DP`, this problem is just one more step further. We iterate through each `word` and see if it can be formed by using other `words`.

Of course it is also obvious that a `word` can only be formed by `words` shorter than it. So we can first sort the input by length of each `word`, and only try to form one `word` by using `words` in front of it.

```
public class Solution {
    public static List<String> findAllConcatenatedWordsInADict(String[] words) {
        List<String> result = new ArrayList<>();
        Set<String> preWords = new HashSet<>();
        Arrays.sort(words, new Comparator<String>() {
            public int compare (String s1, String s2) {
                return s1.length() - s2.length();
            }
        });

        for (int i = 0; i < words.length; i++) {
            if (canForm(words[i], preWords)) {
                result.add(words[i]);
            }
            preWords.add(words[i]);
        }

        return result;
    }

    private static boolean canForm(String word, Set<String> dict) {
        if (dict.isEmpty()) return false;
 boolean[] dp = new boolean[word.length() + 1];
 dp[0] = true;
 for (int i = 1; i <= word.length(); i++) {
     for (int j = 0; j < i; j++) {
  if (!dp[j]) continue;
  if (dict.contains(word.substring(j, i))) {
      dp[i] = true;
      break;
  }
     }
 }
 return dp[word.length()];
    }
}
```

written by shawngao original link here

Solution 2

Most of lines are adding words into Trie Tree
This solution is like putting two pointers to search through the tree. When find a word, put the other pointer back on root then continue searching.
But I'm not sure about the time complexity of my solution. Suppose word length is len and there are n words. Is the time complexity O(len * n ^ 2)?

```
public class Solution {
    class TrieNode {
        TrieNode[] children;
        String word;
        boolean isEnd;
        boolean combo; //if this word is a combination of simple words
        boolean added; //if this word is already added in result
        public TrieNode() {
            this.children = new TrieNode[26];
            this.word = new String();
            this.isEnd = false;
            this.combo = false;
            this.added = false;
        }
    }
    private void addWord(String str) {
        TrieNode node = root;
        for (char ch : str.toCharArray()) {
            if (node.children[ch - 'a'] == null) {
                node.children[ch - 'a'] = new TrieNode();
            }
            node = node.children[ch - 'a'];
        }
        node.isEnd = true;
        node.word = str;
    }
    private TrieNode root;
    private List<String> result;
    public List<String> findAllConcatenatedWordsInADict(String[] words) {
        root = new TrieNode();
        for (String str : words) {
            if (str.length() == 0) {
                continue;
            }
            addWord(str);
        }
        result = new ArrayList<>();
        dfs(root, 0);
        return result;
    }
    private void dfs(TrieNode node, int multi) {
     //multi counts how many single words combined in this word
        if(node.isEnd && !node.added && multi > 1) {
            node.combo = true;
            node.added = true;
            result.add(node.word);
        }
        searchWord(node, root, multi);
```

```
        searchWord(node, root, multi);
    }
    private void searchWord(TrieNode node1, TrieNode node2, int multi) {
        if (node2.combo) {
            return;
        }
        if (node2.isEnd) {
            //take the pointer of node2 back to root
            dfs(node1, multi + 1);
        }
        for (int  i = 0; i < 26; i++) {
            if (node1.children[i] != null && node2.children[i] != null) {
                searchWord(node1.children[i], node2.children[i], multi);
            }
        }
    }
}
```

written by huiyuyang original link here

Solution 3

Let's discuss whether a word should be included in our answer.
Consider the word as a topologically sorted directed graph, where each node is a letter, and an edge exists from i to j if word[i:j] is in our wordlist, [and there is no edge from i=0 to j=len(word)-1]. We want to know if there is a path from beginning to end. If there is, then the word can be broken into concatenated parts from our wordlist. To answer this question, we DFS over this graph.

Code:

```
S = set(A)
ans = []
for word in A:
  if not word: continue
  stack = [0]
  seen = {0}
  M = len(word)
  while stack:
    node = stack.pop()
    if node == M:
      ans.append(word)
      break
    for j in xrange(M - node + 1):
      if (word[node:node+j] in S and
          node + j not in seen and
          (node > 0 or node + j != M)):
        stack.append(node + j)
        seen.add(node + j)

return ans
```

written by awice original link here

From Leetcoder.

Matchsticks to Square

Remember the story of Little Match Girl? By now, you know exactly what matchsticks the little match girl has, please find out a way you can make one square by using up all those matchsticks. You should not break any stick, but you can link them up, and each matchstick must be used **exactly** one time.

Your input will be several matchsticks the girl has, represented with their stick length. Your output will either be true or false, to represent whether you could make one square using all the matchsticks the little match girl has.

Example 1:

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

Explanation: You can form a square with length 2, one side of the square came two s
ticks with length 1.
```

Example 2:

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

Explanation: You cannot find a way to form a square with all the matchsticks.
```

Note:

1. The length sum of the given matchsticks is in the range of `0` to `10^9`.
2. The length of the given matchstick array will not exceed `15`.

Solution 1

According to https://en.wikipedia.org/wiki/Partition_problem, the partition problem (or number partitioning) is the task of deciding whether a given multiset `S` of positive integers can be partitioned into two subsets `S1` and `S2` such that the sum of the numbers in `S1` equals the sum of the numbers in `S2`. The partition problem is `NP-complete`.

When I trying to think how to apply dynamic programming solution of above problem to this one (difference is divid `S` into 4 subsets), I took another look at the constraints of the problem:
The length sum of the given matchsticks is in the range of `0` to `10^9`.
The length of the given matchstick array will not exceed `15`.

Sounds like the input will not be very large... Then why not just do DFS? In fact, DFS solution passed judges.

Anyone solved this problem by using DP? Please let me know :)

```
public class Solution {
    public boolean makesquare(int[] nums) {
     if (nums == null || nums.length < 4) return false;
        int sum = 0;
        for (int num : nums) sum += num;
        if (sum % 4 != 0) return false;

     return dfs(nums, new int[4], 0, sum / 4);
    }

    private boolean dfs(int[] nums, int[] sums, int index, int target) {
     if (index == nums.length) {
         if (sums[0] == target && sums[1] == target && sums[2] == target) {
      return true;
         }
         return false;
     }

     for (int i = 0; i < 4; i++) {
         if (sums[i] + nums[index] > target) continue;
         sums[i] += nums[index];
            if (dfs(nums, sums, index + 1, target)) return true;
         sums[i] -= nums[index];
     }

     return false;
    }
}
```

`Updates on 12/19/2016` Thanks @benjamin19890721 for pointing out a very good optimization: Sorting the input array `DESC` will make the DFS process run much faster. Reason behind this is we always try to put the next matchstick in the first subset. If there is no solution, trying a longer matchstick first will get to negative conclusion earlier. Following is the updated code. Runtime is improved from more

than 1000ms to around 40ms. A big improvement.

```
public class Solution {
    public boolean makesquare(int[] nums) {
     if (nums == null || nums.length < 4) return false;
        int sum = 0;
        for (int num : nums) sum += num;
        if (sum % 4 != 0) return false;

        Arrays.sort(nums);
        reverse(nums);

     return dfs(nums, new int[4], 0, sum / 4);
    }

    private boolean dfs(int[] nums, int[] sums, int index, int target) {
     if (index == nums.length) {
         if (sums[0] == target && sums[1] == target && sums[2] == target) {
      return true;
         }
         return false;
     }

     for (int i = 0; i < 4; i++) {
         if (sums[i] + nums[index] > target) continue;
         sums[i] += nums[index];
            if (dfs(nums, sums, index + 1, target)) return true;
         sums[i] -= nums[index];
     }

     return false;
    }

    private void reverse(int[] nums) {
        int i = 0, j = nums.length - 1;
        while (i < j) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
            i++; j--;
        }
    }
}
```

written by shawngao original link here

Solution 2

This is a NP problem. Time complexity should be O(4 ^ n), n is the length of array.

Trivial Solution:

```
class Solution {
    bool dfs(vector<int> &sidesLength,const vector<int> &matches, int index) {
        if (index == matches.size())
            return sidesLength[0] == sidesLength[1] && sidesLength[1] == sidesLen
gth[2] && sidesLength[2] == sidesLength[3];
        for (int i = 0; i < 4; ++i) {
            sidesLength[i] += matches[index];
            if (dfs(sidesLength, matches, index + 1))
                return true;
            sidesLength[i] -= matches[index];
        }
        return false;
    }
public:
    bool makesquare(vector<int>& nums) {
        if (nums.empty()) return false;
        vector<int> sidesLength(4, 0);
        return dfs(sidesLength, nums, 0);
    }
};
```

Without pruning, this solution TLEs at 11th test case.

First Optimization:

> each matchstick must be used exactly one time.

The description says we need to use every single match exactly **once**, so we can get the **length of each side of the square** if there is one.
if the current length is larger than target length, we don't need to go any further. `if (sidesLength[i] + matches[index] > target) continue;` by adding this line of code into dfs function, solution get TLE at 147th test case.

Second Optimization:

After reading the description again, I realize that the length of a single match can be very long. If they put long ones after short ones, it will take a long time before my algorithm return false.
So I sort all the matches to avoid the worst case: `sort(nums.begin(), nums.end(), [](const int &l, const int &r){return l > r;});`. After putting this line before my algorithm, solution passed all cases and beats 50% solutions. I saw someone's algorithm is amazingly fast, they even make reconsider wether the problem is NP or not.

Third Optimization:

Because their solutions is so fast, I need think about if I can use DP to solve the

problem. It turns out that it's still a NP problem, which makes happy again. But I can actually use the concept of DP in optimization: **if I have checked the same length before, why do I need to bother checking again?**
Although we only have 4 sides in a square, we can still check if we have encountered the same length with the current match. After I add this to my code:

```
int j = i;
while (--j >= 0)
    if (sidesLength[i] == sidesLength[j])
        break;
if (j != -1) continue;
```

It passed all test case in 6ms.

```
class Solution {
    bool dfs(vector<int> &sidesLength,const vector<int> &matches, int index, cons
t int target) {
        if (index == matches.size())
            return sidesLength[0] == sidesLength[1] && sidesLength[1] == sidesLen
gth[2] && sidesLength[2] == sidesLength[3];
        for (int i = 0; i < 4; ++i) {
            if (sidesLength[i] + matches[index] > target) // first
                continue;
            int j = i;
            while (--j >= 0) // third
                if (sidesLength[i] == sidesLength[j])
                    break;
            if (j != -1) continue;
            sidesLength[i] += matches[index];
            if (dfs(sidesLength, matches, index + 1, target))
                return true;
            sidesLength[i] -= matches[index];
        }
        return false;
    }
public:
    bool makesquare(vector<int>& nums) {
        if (nums.size() < 4) return false;
        int sum = 0;
        for (const int val: nums) {
            sum += val;
        }
        if (sum % 4 != 0) return false;
        sort(nums.begin(), nums.end(), [](const int &l, const int &r){return l >
r;}); // second
        vector<int> sidesLength(4, 0);
        return dfs(sidesLength, nums, 0, sum / 4);
    }
};
```

written by withacup original link here

Solution 3

This is a solution inspired by a friend who doesn't do leetcode. I am just posting his solution with the best explanation I can give. The bitmasking technique may look sophisticated but the idea is actually pretty straightforward because it uses brute force with some optimizations. A bitmask is used as a representation of a subset. For example if nums = {1,1,2,2,2}, then a bitmask = 01100 represents the subset {1,2}.

```
bool makesquare(vector<int>& nums) {
    int n = nums.size();

    long sum = accumulate(nums.begin(), nums.end(), 0l);
    if (sum % 4)
        return false;
    long sideLen = sum / 4;
    // need to solve the problem of partitioning nums into four equal subsets eac
h having
    // sum equal to sideLen
    vector<int> usedMasks;
    // validHalfSubsets[i] == true iff the subset represented by bitmask i
    // has sum == 2*sideLen, AND the subset represented by i can be further parti
tioned into
    // two equal subsets. See below for how it is used.
    vector<bool> validHalfSubsets(1<<n, false);

    // E.g., if n = 5, (1 << 5 - 1) = 11111 represents the whole set
    int all = (1 << n) - 1;
    // go through all possible subsets each represented by a bitmask
    for (int mask = 0; mask <= all; mask++) {
        long subsetSum = 0;
        // calculate the sum of this subset
        for (int i = 0; i < 32; i++) {
     if ((mask >> i) & 1)
  subsetSum += nums[i];
        }
 // if this subset has what we want
 if (subsetSum == sideLen) {
     for (int usedMask : usedMasks) {
     // if this mask and usedMask are mutually exclusive
         if ((usedMask & mask) == 0) {
      // then they form a valid half subset whose sum is 2 * sideLen,
                    // that can be further partitioned into two equal subsets (us
edMask and mask)
      int validHalf = usedMask | mask;
      validHalfSubsets[validHalf] = true;
      // if in the past we concluded that the other half is also a valid
      // half subset, DONE!
      if (validHalfSubsets[all ^ validHalf])
          return true;
         }
            }
     usedMasks.push_back(mask);
        }
    }
    return false;
}
```

written by yangluphil original link here

Ones and Zeroes

In the computer world, use restricted resource you have to generate maximum benefit is what we always want to pursue.

For now, suppose you are a dominator of **m** `0s` and **n** `1s` respectively. On the other hand, there is an array with strings consisting of only `0s` and `1s`.

Now your task is to find the maximum number of strings that you can form with given **m** `0s` and **n** `1s`. Each `0` and `1` can be used at most **once**.

Note:

1. The given numbers of `0s` and `1s` will both not exceed `100`
2. The size of given string array won't exceed `600`.

Example 1:

```
Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3
Output: 4

Explanation: This are totally 4 strings can be formed by the using of 5 0s and 3 1s
, which are “10,”0001”,”1”,”0”
```

Example 2:

```
Input: Array = {"10", "0", "1"}, m = 1, n = 1
Output: 2

Explanation: You could form "10", but then you'd have nothing left. Better form "0"
 and "1".
```

Solution 1

```
int findMaxForm(vector<string>& strs, int m, int n) {
  vector<vector<int>> memo(m+1, vector<int>(n+1, 0));
  int numZeroes, numOnes;

  for (auto &s : strs) {
    numZeroes = numOnes = 0;
    // count number of zeroes and ones in current string
    for (auto c : s) {
      if (c == '0')
 numZeroes++;
      else if (c == '1')
 numOnes++;
    }

    // memo[i][j] = the max number of strings that can be formed with i 0's and j
1's
    // from the first few strings up to the current string s
    // Catch: have to go from bottom right to top left
    // Why? If a cell in the memo is updated(because s is selected),
    // we should be adding 1 to memo[i][j] from the previous iteration (when we we
re not considering s)
    // If we go from top left to bottom right, we would be using results from this
iteration => overcounting
    for (int i = m; i >= numZeroes; i--) {
 for (int j = n; j >= numOnes; j--) {
          memo[i][j] = max(memo[i][j], memo[i - numZeroes][j - numOnes] + 1);
 }
    }
  }
  return memo[m][n];
}
```

written by yangluphil original link here

Solution 2

Time Complexity: O(kl + kmn), where k is the length of input string array and l is the average length of a string within the array.

```
public int findMaxForm(String[] strs, int m, int n) {
    int[][] dp = new int[m+1][n+1];
    for (String s : strs) {
        int[] count = count(s);
        for (int i=m;i>=count[0];i--)
            for (int j=n;j>=count[1];j--)
                dp[i][j] = Math.max(1 + dp[i-count[0]][j-count[1]], dp[i][j]);
    }
    return dp[m][n];
}

public int[] count(String str) {
    int[] res = new int[2];
    for (int i=0;i<str.length();i++)
        res[str.charAt(i) - '0']++;
    return res;
 }
```

Thanks @shawngao for some ways to make this solution more concise.

written by compton_scatter original link here

Solution 3

This question is very similar to a 0-1 knapsack, the transition function is

```
dp(k, x, y) = max(dp(k-1, x-z, y-o) + 1, dp(k-1, x, y))   (z is zeroes in strs[k]
, o is ones in strs[k])
```

dp(k, x, y) is the maximum strs we can include when we have x zeros, y ones and only the first k strs are considered.

dp(len(strs), M, N) is the answer we are looking for

I first implemented a dfs + memoization, which gets MLE, so I created a bottom up style dp.
With bottom up, we can use something called "rolling array" to optimize space complexity from O(KMN) to O(MN)

```
class Solution(object):
    def findMaxForm(self, strs, m, n):

        dp = [[0] * (n+1) for _ in range(m+1)]

        def count(s):
            return sum(1 for c in s if c == '0'), sum(1 for c in s if c == '1')

        for z, o in [count(s) for s in strs]:
            for x in range(m, -1, -1):
                for y in range(n, -1, -1):
                    if x >= z and y >= o:
                        dp[x][y] = max(1 + dp[x-z][y-o], dp[x][y])

        return dp[m][n]
```

written by mlawo original link here

Heaters

Winter is coming! Your first job during the contest is to design a standard heater with fixed warm radius to warm all the houses.

Now, you are given positions of houses and heaters on a horizontal line, find out minimum radius of heaters so that all houses could be covered by those heaters.

So, your input will be the positions of houses and heaters seperately, and your expected output will be the minimum radius standard of heaters.

Note:

1. Numbers of houses and heaters you are given are non-negative and will not exceed 25000.
2. Positions of houses and heaters you are given are non-negative and will not exceed 10^9.
3. As long as a house is in the heaters' warm radius range, it can be warmed.
4. All the heaters follow your radius standard and the warm radius will the same.

Example 1:

```
Input: [1,2,3],[2]
Output: 1
Explanation: The only heater was placed in the position 2, and if we use the radius
 1 standard, then all the houses can be warmed.
```

Example 2:

```
Input: [1,2,3,4],[1,4]
Output: 1
Explanation: The two heater was placed in the position 1 and 4. We need to use radi
us 1 standard, then all the houses can be warmed.
```

Solution 1

The idea is to leverage decent `Arrays.binarySearch()` function provided by Java.

1. For each `house`, find its position between those `heaters` (thus we need the `heaters` array to be sorted).
2. Calculate the distances between this `house` and left `heater` and right `heater`, get a `MIN` value of those two values. Corner cases are there is no left or right heater.
3. Get `MAX` value among distances in step 2. It's the answer.

Time complexity: max(O(nlogn), O(mlogn)) - m is the length of houses, n is the length of heaters.

```
public class Solution {
    public int findRadius(int[] houses, int[] heaters) {
        Arrays.sort(heaters);
        int result = Integer.MIN_VALUE;

        for (int house : houses) {
         int index = Arrays.binarySearch(heaters, house);
         if (index < 0) {
          index = -(index + 1);
         }
         int dist1 = index - 1 >= 0 ? house - heaters[index - 1] : Integer.MAX_VA
LUE;
         int dist2 = index < heaters.length ? heaters[index] - house : Integer.MA
X_VALUE;

         result = Math.max(result, Math.min(dist1, dist2));
        }

        return result;
    }
}
```

written by shawngao original link here

Solution 2

Based on 2 pointers, the idea is to find the nearest heater for each house, by comparing the next heater with the current heater.

```
public class Solution {
    public int findRadius(int[] houses, int[] heaters) {
        Arrays.sort(houses);
        Arrays.sort(heaters);

        int i = 0, j = 0, res = 0;
        while (i < houses.length) {
            while (j < heaters.length - 1
                && Math.abs(heaters[j + 1] - houses[i]) <= Math.abs(heaters[j] -
houses[i])) {
                j++;
            }
            res = Math.max(res, Math.abs(heaters[j] - houses[i]));
            i++;
        }

        return res;
    }
}
```

Updated solution inspired by @StefanPochmann

```
public class Solution {
    public int findRadius(int[] houses, int[] heaters) {
        Arrays.sort(houses);
        Arrays.sort(heaters);

        int i = 0, res = 0;
        for (int house : houses) {
            while (i < heaters.length - 1 && heaters[i] + heaters[i + 1] <= house
* 2) {
                i++;
            }
            res = Math.max(res, Math.abs(heaters[i] - house));
        }

        return res;
    }
}
```

written by lufangjianle original link here

Solution 3

Go through houses and heaters in ascending order. My `i` points to the current closest heater. Go to the next heater if the current house coordinate is larger than or equal to the middle between the current and the next heater.

```
def findRadius(self, houses, heaters):
    heaters = sorted(heaters) + [float('inf')]
    i = r = 0
    for x in sorted(houses):
        while x >= sum(heaters[i:i+2]) / 2.:
            i += 1
        r = max(r, abs(heaters[i] - x))
    return r
```

I btw started with
`while abs(heaters[i+1] - x) <= abs(heaters[i] - x):`,
the straight-forward check whether the next heater is closer than the current. Then I thought I probably don't need `abs` if I just use
`while heaters[i+1] - x <= x - heaters[i]:`.
That's obviously correct if `x` is between the heaters, because then that's the correct distances of `x` to the two heaters. Less obviously (but imho not surprisingly) it's also correct if `x` isn't between them. Finally, after rewriting it to
`while heaters[i] + heaters[i+1] <= 2 * x:`
I realized what that meant :-)

Update: Another solution by using binary search, inspired by others:

```
def findRadius(self, houses, heaters):
    heaters.sort()
    return max(min(abs(house - heater)
                   for i in [bisect.bisect(heaters, house)]
                   for heater in heaters[i-(i>0):i+1])
               for house in houses)
```

written by StefanPochmann original link here

From Leetcoder.

Number Complement

Given a positive integer, output its complement number. The complement strategy is to flip the bits of its binary representation.

Note:

1. The given integer is guaranteed to fit within the range of a 32-bit signed integer.
2. You could assume no leading zero bit in the integer's binary representation.

Example 1:

```
Input: 5
Output: 2
Explanation: The binary representation of 5 is 101 (no leading zero bits), and its
complement is 010. So you need to output 2.
```

Example 2:

```
Input: 1
Output: 0
Explanation: The binary representation of 1 is 1 (no leading zero bits), and its co
mplement is 0. So you need to output 0.
```

Solution 1

```
class Solution {
public:
    int findComplement(int num) {
        unsigned mask = ~0;
        while (num & mask) mask <<= 1;
        return ~mask & ~num;
    }
};
```

For example,

```
num          = 00000101
mask         = 11111000
~mask & ~num = 00000010
```

written by lzl124631x137 original link here

Solution 2

I post solution first and then give out explanation. Please think why does it work before read my explanation.

```
public class Solution {
    public int findComplement(int num) {
        return ~num & ((Integer.highestOneBit(num) << 1) - 1);
    }
}
```

According to the problem, the result is

1. The `flipped` version of the original `input` but
2. Only flip `N` bits within the range from `LEFTMOST` bit of `1` to `RIGHTMOST`. For example input = `5` (the binary representation is `101`), the `LEFTMOST` bit of `1` is the third one from `RIGHTMOST` (`100`, `N` = 3). Then we need to flip 3 bits from `RIGHTMOST` and the answer is `010`

To achieve above algorithm, we need to do 3 steps:

1. Create a bit mask which has `N` bits of `1` from `RIGHTMOST`. In above example, the mask is `111`. And we can use the decent Java built-in function `Integer.highestOneBit` to get the `LEFTMOST` bit of `1`, left shift one, and then minus one. Please remember this wonderful trick to create bit masks with `N` ones at `RIGHTMOST`, you will be able to use it later.
2. Negate the whole input number.
3. `Bit AND` numbers in step `1` and `2`.

Three line solution if you think one line solution is too confusing:

```
public class Solution {
    public int findComplement(int num) {
        int mask = (Integer.highestOneBit(num) << 1) - 1;
        num = ~num;
        return num & mask;
    }
}
```

written by shawngao original link here

Solution 3

```
class Solution(object):
    def findComplement(self, num):
        i = 1
        while i <= num:
            i = i << 1
        return (i - 1) ^ num
```

written by Ipeq1 original link here

From Leetcoder.

Total Hamming Distance

The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Now your job is to find the total Hamming distance between all pairs of the given numbers.

Example:

```
Input: 4, 14, 2

Output: 6

Explanation: In binary representation, the 4 is 0100, 14 is 1110, and 2 is 0010 (ju
st
showing the four bits relevant in this case). So the answer will be:
HammingDistance(4, 14) + HammingDistance(4, 2) + HammingDistance(14, 2) = 2 + 2 + 2
 = 6.
```

Note:

1. Elements of the given array are in the range of `0` to `10^9`
2. Length of the array will not exceed `10^4`.

Solution 1

For each bit position 1-32 in a 32-bit integer, we count the number of integers in the array which have that bit set. Then, if there are n integers in the array and k of them have a particular bit set and (n-k) do not, then that bit contributes k*(n-k) hamming distance to the total.

```
public int totalHammingDistance(int[] nums) {
    int total = 0, n = nums.length;
    for (int j=0;j<32;j++) {
        int bitCount = 0;
        for (int i=0;i<n;i++)
            bitCount += (nums[i] >> j) & 1;
        total += bitCount*(n - bitCount);
    }
    return total;
}
```

written by compton_scatter original link here

Solution 2

```
def totalHammingDistance(self, nums):
    return sum(b.count('0') * b.count('1') for b in zip(*map('{:032b}'.format, nu
ms)))
```

written by StefanPochmann original link here

Solution 3

1. Problem

The problem is to find the total Hamming distance between all pairs of the given numbers.

2. Thinking process

2.1 For one pair

When you calculate Hamming distance between x and y, you just

1. calculate p = x ^ y;
2. count the number of 1's in p

The distance from x to y is as same as y to x.

2.2 Trival approach

For a series of number: a1, a2, a3,..., an

Use the approach in 2.1
(suppose distance(x, y) is the Hamming distance between x and y):

For a1, calculate S(1) = distance(a1, a2)+distance(a1, a3)+ ... +distance(a1, an)
For a2, calculate S(2) = distance(a2, a3)+distance(a2, a4)+ ... +distance(a2, an)
.....
For a(n - 1), calculate S(n - 1) = distance(a(n - 1), a(n))

Finally , **S = S(1) + S(2) + ... + S(n - 1)** .

The function distance is called **1 + 2 + ... + (n - 1) = n(n - 1)/2** times! That's too much!

2.3 New idea

The total Hamming distance is constructed **bit by bit** in this approach.

Let's take a series of number: **a1, a2, a3,..., an**

Just think about all the **Least Significant Bit (LSB)** of a(k) (1 ≤ k ≤ n).

How many Hamming distance will they bring to the total?

1. If a pair of number has same LSB, the total distance will get 0.
2. If a pair of number has different LSB, the total distance will get 1.

For all number **a1, a2, a3,..., a(n)**, if there are **p** numbers have **0 as LSB (put in set M)**, and **q** numbers have **1 for LSB (put in set N)** .

There are **2 situations**:

Situation 1. If the **2 number in a pair both comes from M (or N)** , the total will get **0**.

Situation 2. If the **1 number in a pair comes from M, the other comes from N**, the total will get **1**.

Since **Situation 1** will add **NOTHING** to the total, we only need to think about **Situation 2**

How many pairs are there in Situation 2?

We choose **1 number from M (p possibilities)** , and **1 number from N (q possibilities)**.

The total possibilities is **p × q = pq** , which means

> **The total Hamming distance will get pq from LSB.**

If we **remove the LSB of all numbers (right logical shift)** , the same idea can be used **again and again until all numbers becomes zero**

2.4 Time Complexity

In each loop, we need to **visit all numbers in nums** to **calculate how many numbers has 0 (or 1) as LSB**.

If the biggest number in nums[] is K, **the total number of loop is [logM] + 1** .

So, **the total time complexity of this approach is O(n)** .

---

3. Code

```
class Solution {
public:
    int totalHammingDistance(vector<int>& nums) {
        int size = nums.size();
        if(size < 2) return 0;
        int ans = 0;
        int *zeroOne = new int[2];
        while(true)
        {
            int zeroCount = 0;
            zeroOne[0] = 0;
            zeroOne[1] = 0;
            for(int i = 0; i < nums.size(); i++)
            {
                if(nums[i] == 0) zeroCount++;
                zeroOne[nums[i] % 2]++;
                nums[i] = nums[i] >> 1;
            }
            ans += zeroOne[0] * zeroOne[1];
            if(zeroCount == nums.size()) return ans;
        }
    }
};
```

---

written by KJer original link here

From Leetcoder.

Sliding Window Median

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Examples:
`[2,3,4]` , the median is `3`

`[2,3]`, the median is `(2 + 3) / 2 = 2.5`

Given an array *nums*, there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position. Your job is to output the median array for each window in the original array.

For example,
Given *nums* = `[1,3,-1,-3,5,3,6,7]`, and *k* = 3.

```
Window position                Median
---------------               -----
[1  3  -1] -3  5  3  6  7       1
 1 [3  -1  -3] 5  3  6  7       -1
 1  3 [-1  -3  5] 3  6  7       -1
 1  3  -1 [-3  5  3] 6  7       3
 1  3  -1  -3 [5  3  6] 7       5
 1  3  -1  -3  5 [3  6  7]      6
```

Therefore, return the median sliding window as `[1,-1,-1,3,5,6]`.

Note:
You may assume *k* is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.

Solution 1

Keep the window elements in a multiset and keep an iterator pointing to the middle value (to "index" k/2, to be precise). Thanks to @votrubac's solution and comments.

```
vector<double> medianSlidingWindow(vector<int>& nums, int k) {
    multiset<int> window(nums.begin(), nums.begin() + k);
    auto mid = next(window.begin(), k / 2);
    vector<double> medians;
    for (int i=k; ; i++) {

        // Push the current median.
        medians.push_back((double(*mid) + *next(mid, k%2 - 1)) / 2);

        // If all done, return.
        if (i == nums.size())
            return medians;

        // Insert nums[i].
        window.insert(nums[i]);
        if (nums[i] < *mid)
            mid--;

        // Erase nums[i-k].
        if (nums[i-k] <= *mid)
            mid++;
        window.erase(window.lower_bound(nums[i-k]));
    }
}
```

written by StefanPochmann original link here

Solution 2

There are a few solutions using BST with worst case time complexity O(n*k), but we know k can be become large. I wanted to come up with a solution that is guaranteed to run in O(n*log(n)) time. This is in my opinion the best solution so far.

The idea is inspired by solutions to Find Median from Data Stream: use two heaps to store numbers in the sliding window. However there is the issue of numbers moving out of the window, and it turns out that a hash table that records these numbers will just work (and is surprisingly neat). The recorded numbers will only be deleted when they come to the top of the heaps.

```
class Solution {
public:
    vector<double> medianSlidingWindow(vector<int>& nums, int k) {
        vector<double> medians;
        unordered_map<int, int> hash;                          // count numbers
to be deleted
        priority_queue<int, vector<int>> bheap;                // heap on the bot
tom
        priority_queue<int, vector<int>, greater<int>> theap;  // heap on the top

        int i = 0;

        // Initialize the heaps
        while (i < k)  { bheap.push(nums[i++]); }
        for (int count = k/2; count > 0; --count) {
            theap.push(bheap.top()); bheap.pop();
        }

        while (true) {
            // Get median
            if (k % 2) medians.push_back(bheap.top());
            else medians.push_back( ((double)bheap.top() + theap.top()) / 2 );

            if (i == nums.size()) break;
            int m = nums[i-k], n = nums[i++], balance = 0;

            // What happens to the number m that is moving out of the window
            if (m <= bheap.top())  { --balance;  if (m == bheap.top()) bheap.pop(
); else ++hash[m]; }
            else                   { ++balance;  if (m == theap.top()) theap.pop(
); else ++hash[m]; }

            // Insert the new number n that enters the window
            if (!bheap.empty() && n <= bheap.top())  { ++balance; bheap.push(n);
}
            else                                     { --balance; theap.push(n);
}

            // Rebalance the bottom and top heaps
            if      (balance < 0)  { bheap.push(theap.top()); theap.pop(); }
            else if (balance > 0)  { theap.push(bheap.top()); bheap.pop(); }

            // Remove numbers that should be discarded at the top of the two heap
s
            while (!bheap.empty() && hash[bheap.top()])  { --hash[bheap.top()]; b
heap.pop(); }
            while (!theap.empty() && hash[theap.top()])  { --hash[theap.top()]; t
heap.pop(); }
        }

        return medians;
    }
};
```

Since both heaps will never have a size greater than n, the time complexity is

O(n*log(n)) in the worst case.

written by ipt original link here

Solution 3

The idea is to maintain a BST of the window and just search for the k/2 largest element and k/2 smallest element then the average of these two is the median of the window.

Now if the STL's multiset BST maintained how many element were in each subtree finding each median would take O(log k) time but since it doesn't it takes O(k) time to find each median.

```
public:
    vector<double> medianSlidingWindow(vector<int>& nums, int k) {
        multiset<int> mp;
        vector<double> med;

        for(int i=0; i<k-1; ++i) mp.insert(nums[i]);

        for(int i=k-1; i< nums.size(); ++i){
            mp.insert(nums[i]); // Add the next number

            auto itb = mp.begin(); advance(itb, (k-1)/2); //Find the lower median
            auto ite = mp.end(); advance(ite, -(k+1)/2); //Find the upper median

            double avg = ((long)(*itb) + (*ite)) / 2.0;
            med.push_back(avg);

            mp.erase(mp.find(nums[i-k+1])); //Remove the oldest element
        }

        return med;
    }
};
```

written by kevin36 original link here

From Leetcoder.

Magical String

A magical string **S** consists of only '1' and '2' and obeys the following rules:

The string **S** is magical because concatenating the number of contiguous occurrences of characters '1' and '2' generates the string **S** itself.

The first few elements of string **S** is the following: **S** = "1221121221221121122……"

If we group the consecutive '1's and '2's in **S**, it will be:

1 22 11 2 1 22 1 22 11 2 11 22 ……

and the occurrences of '1's or '2's in each group are:

1 2 2 1 1 2 1 2 2 1 2 2 ……

You can see that the occurrence sequence above is the **S** itself.

Given an integer N as input, return the number of '1's in the first N number in the magical string **S**.

Note: N will not exceed 100,000.

Example 1:

```
Input: 6
Output: 3
Explanation: The first 6 elements of magical string S is "12211" and it contains th
ree 1's, so return 3.
```

Solution 1

Algorithm:

1. Create an `int` array `a` and initialize the first 3 elements with `1, 2, 2`.
2. Create two pointers `head` and `tail`. `head` points to the number which will be used to generate new numbers. `tail` points to the next empty position to put the new number. Then keep generating new numbers until `tail` >= `n`.
3. Need to create the array 1 element more than `n` to avoid overflow because the last round `head` might points to a number `2`.
4. A trick to flip number back and forth between `1` and `2`: `num = num ^ 3`

```
public class Solution {
    public int magicalString(int n) {
        if (n <= 0) return 0;
        if (n <= 3) return 1;

        int[] a = new int[n + 1];
        a[0] = 1; a[1] = 2; a[2] = 2;
        int head = 2, tail = 3, num = 1, result = 1;

        while (tail < n) {
            for (int i = 0; i < a[head]; i++) {
                a[tail] = num;
                if (num == 1 && tail < n) result++;
                tail++;
            }
            num = num ^ 3;
            head++;
        }

        return result;
    }
}
```

written by shawngao original link here

Solution 2

Just build enough of the string and then count.

```
int magicalString(int n) {
    string S = "122";
    int i = 2;
    while (S.size() < n)
        S += string(S[i++] - '0', S.back() ^ 3);
    return count(S.begin(), S.begin() + n, '1');
}
```

written by StefanPochmann original link here

Solution 3

```
public int magicalString(int n) {
        StringBuilder magic = new StringBuilder("1221121221221121122");
        int pt1 = 12, pt2 = magic.length(), count = 0; //initiate pointers
        //generate sequence directly
        while(magic.length() < n){
            if(magic.charAt(pt1) == '1'){
                if(magic.charAt(pt2-1) == '1') magic.append(2);
                else magic.append(1);
                pt2++;
            }else{ //==2
                if(magic.charAt(pt2-1) == '1') magic.append(22);
                else magic.append(11);
                pt2+=2;
            }
            pt1++;
        }
        for(int i=0;i<n;i++)
            if(magic.charAt(i)=='1') count++;
        return count;
    }
```

written by cgxy1995 original link here

From Leetcoder.

License Key Formatting

Now you are given a string S, which represents a software license key which we would like to format. The string S is composed of alphanumerical characters and dashes. The dashes split the alphanumerical characters within the string into groups. (i.e. if there are M dashes, the string is split into M+1 groups). The dashes in the given string are possibly misplaced.

We want each group of characters to be of length K (except for possibly the first group, which could be shorter, but still must contain at least one character). To satisfy this requirement, we will reinsert dashes. Additionally, all the lower case letters in the string must be converted to upper case.

So, you are given a non-empty string S, representing a license key to format, and an integer K. And you need to return the license key formatted according to the description above.

Example 1:

```
Input: S = "2-4A0r7-4k", K = 4

Output: "24A0-R74K"

Explanation: The string S has been split into two parts, each part has 4 characters
.
```

Example 2:

```
Input: S = "2-4A0r7-4k", K = 3

Output: "24-A0R-74K"

Explanation: The string S has been split into three parts, each part has 3 characte
rs except the first part as it could be shorter as said above.
```

Note:

1. The length of string S will not exceed 12,000, and K is a positive integer.
2. String S consists only of alphanumerical characters (a-z and/or A-Z and/or 0-9) and dashes(-).
3. String S is non-empty.

Solution 1

```
    public String licenseKeyFormatting(String s, int k) {
        StringBuilder sb = new StringBuilder();
        for (int i = s.length() - 1; i >= 0; i--)
            if (s.charAt(i) != '-')
                sb.append(sb.length() % (k + 1) == k ? '-' : "").append(s.charAt(
i));
        return sb.reverse().toString().toUpperCase();
    }
```

written by yuxiangmusic original link here

Solution 2

```
class Solution(object):
    def licenseKeyFormatting(self, S, K):
        """
        :type S: str
        :type K: int
        :rtype: str
        """
        S = S.upper().replace('-','')
        size = len(S)
        s1 = K if size%K==0 else size%K
        res = S[:s1]
        while s1<size:
            res += '-'+S[s1:s1+K]
            s1 += K
        return res
```

written by yujia003gmail.com original link here

Solution 3

```java
public class Solution {
    public String licenseKeyFormatting(String S, int K) {
        String[] strs = S.split("-");
        StringBuilder rst = new StringBuilder();
        int gap = 'A' - 'a';

        for (String str : strs) {
            rst.append(str);
        }

        int len = rst.length();

        for (int i = 0; i < len; i++) {
            char c = rst.charAt(i);
            if (c>='a' && c<='z') {
                rst.setCharAt(i, (char)(c+gap));
            }
        }

        for (int i = len-K; i > 0; i -= K) {
            rst.insert(i, '-');
        }
        return rst.toString();
    }
}
```

written by zjay original link here

From Leetcoder.

Smallest Good Base

For an integer n, we call k>=2 a ***good base*** of n, if all digits of n base k are 1.

Now given a string representing n, you should return the smallest good base of n in string format.

Example 1:

```
Input: "13"
Output: "3"
Explanation: 13 base 3 is 111.
```

Example 2:

```
Input: "4681"
Output: "8"
Explanation: 4681 base 8 is 11111.
```

Example 3:

```
Input: "1000000000000000000"
Output: "999999999999999999"
Explanation: 1000000000000000000 base 999999999999999999 is 11.
```

Note:

1. The range of n is [3, 10^18].
2. The string representing n is always valid and will not have leading zeros.

Solution 1

```
class Solution(object):
    def smallestGoodBase(self, N):
        n = int(N);

        for k in xrange(int(math.log(n, 2)), 1 , -1):
            a = int(n ** k ** -1)                    # kth-root of n
            if (1 - a ** (k + 1)) // (1 - a) == n: # [a^0 + a^1 + ... + a^k] == n
                return str(a)

        return str(n - 1)
```

written by hausch original link here

Solution 2

First things first. Let's see the math behind it.

From given information, we can say one thing- Numbers will be of form-

n = k^m + k^(m-1) + ... + k + 1
=> n-1 = k^m + k^(m-1) + ... + k
=> n-1 = k (k^(m-1) + k^(m-2) + ... + k + 1) ..... **[1]**

Also, from n = k^m + k^(m-1) + ... + k + 1, we can say,
n-k^m = k^(m-1) + k^(m-2) + ... + k + 1 ..... **[2]**

from [1] and [2],

n-1 = k (n - k^m)
=>k^(m+1) = nk - n + 1

if you shuffle sides you will end up getting following form,

(k^(m+1) - 1)/(k - 1) = n

Also from [1] note that, (n - 1) must be divisible by k.**[3]**

[EDIT] -->

With inputs from @StefanPochmann we can also say, from binomial thorem, **n = k^m + ... + 1 < (k+1)^m**, therefore, k+1 > m-th root of n > k. Thus ⌊**m-th root of n**⌋ is the only candidate that needs to be tested.**[4]**

<--

So our number should satisfy this equation where **k** will be our base and **m** will be (number of 1s - 1)

This brings us to the search problem where we need to find k and m.

Linear search from 1 to n does not work. it gives us TLE. So it leaves us with performing some optimization on search space.

From **[4]** we know that the only candidate that needs to be tested is,⌊**m-th root of n**⌋

We also know that the smallest base is 2 so we can find our m must be between 2 and $\log_2 n$ else m is (n-1) **[5]**

That brings me to the code:
[EDIT] -- >

```python
import math
class Solution(object):
    def smallestGoodBase(self, n):
        """
        :type n: str
        :rtype: str
        """
        n = int(n)
        max_m = int(math.log(n,2))
        for m in range(max_m,1,-1):
            k = int(n**m**-1)
            if (k**(m+1)-1)//(k-1) == n:
                return str(k)

        return str(n-1)
```

<—-

written by harshaneel original link here

Solution 3

sum base r == 1111... means sum = 1+r+r^2+r^3+...+r^p
I tried two different search.

1. for r = 2-10, increase p until sum>= n
2. for p=2,3,4..., calculate r = sum^(1/p). then try a few numbers around r to see if the sum fits the above equation. In fact, I only tried r and got accepted by OJ.

```
public String smallestGoodBase(String n) {
    long nl = 0, cur = 1;
    for (int i=n.length()-1;i>=0;i--){
        nl+=(n.charAt(i)-'0')*cur;
        cur*=10;
    }
    for (long i=2;i<10;i++){
        long s = 0;
        cur = 1;
        for (int j=0;j<nl;j++){
            s+=cur;
            cur*=i;
            if (s == nl) return Long.toString(i);
            if (s > nl) break;
        }
    }
    long res = nl-1;
    for (int i=2;i<1000;i++){
     int r = (int)Math.pow(nl,1.0/i);
     if (r<5) break;
     if (helper(r,i,nl)&&res>r)
      res = r;
    }
    return Long.toString(res);
}
boolean helper(int r, int i, long nl){
 long res = 0;
 long cur = 1;
 for(int j=0;j<=i;j++){
  res+=cur;
  cur*=r;
  if (cur>1000000000)
   cur%=1000000000;
 }
 if (res%1000000000 == nl%1000000000) return true;
 else return false;
}
```

written by jinsheng original link here

From Leetcoder.

Find Permutation

By now, you are given a **secret signature** consisting of character 'D' and 'I'. 'D' represents a decreasing relationship between two numbers, 'I' represents an increasing relationship between two numbers. And our **secret signature** was constructed by a special integer array, which contains uniquely all the different number from 1 to n (n is the length of the secret signature plus 1). For example, the secret signature "DI" can be constructed by array [2,1,3] or [3,1,2], but won't be constructed by array [3,2,4] or [2,1,3,4], which are both illegal constructing special string that can't represent the "DI" **secret signature**.

On the other hand, now your job is to find the lexicographically smallest permutation of [1, 2, ... n] could refer to the given **secret signature** in the input.

Example 1:

```
Input: "I"
Output: [1,2]
Explanation: [1,2] is the only legal initial spectial string can construct secret s
ignature "I", where the number 1 and 2 construct an increasing relationship.
```

Example 2:

```
Input: "DI"
Output: [2,1,3]
Explanation: Both [2,1,3] and [3,1,2] can construct the secret signature "DI", but
since we want to find the one with the smallest lexicographical permutation, you ne
ed to output [2,1,3]
```

Note:

- The input string will only contain the character 'D' and 'I'.
- The length of input string is a positive integer and will not exceed 10,000

Solution 1

I have used a greedy algorithm:

1. Loop on the input and insert a decreasing numbers when see a 'I'
2. Insert a decreasing numbers to complete the result.

Simple example:
Input: "DIDDID"
0 []
1 [2, 1]
2 [2, 1]
3 [2, 1]
4 [2, 1, 5, 4, 3]
5 [2, 1, 5, 4, 3]
[2, 1, 5, 4, 3, 7, 6]

Then, output is [2, 1, 5, 4, 3, 7, 6]

```
def findPermutation(self, s):
    ret = []
    for i in range(len(s)):
      if s[i] == 'I':
        ret.extend(range(i + 1, len(ret), -1))
    ret.extend(range(len(s) + 1, len(ret), -1))
    return ret
```

Hope that you like my solution. :)

written by lee215 original link here

Solution 2

For example, given `IDIIDD` we start with sorted sequence `1234567`
Then for each `k` continuous `D` starting at index `i` we need to reverse `[i, i+k]` portion of the sorted sequence.

```
IDIIDD
1234567 // sorted
1324765 // answer
```

```
    public int[] findPermutation(String s) {
        int n = s.length(), arr[] = new int[n + 1];
        for (int i = 0; i <= n; i++) arr[i] = i + 1; // sorted
        for (int h = 0; h < n; h++) {
            if (s.charAt(h) == 'D') {
                int l = h;
                while (h < n && s.charAt(h) == 'D') h++;
                reverse(arr, l, h);
            }
        }
        return arr;
    }

    void reverse(int[] arr, int l, int h) {
        while (l < h) {
            arr[l] ^= arr[h];
            arr[h] ^= arr[l];
            arr[l] ^= arr[h];
            l++; h--;
        }
    }
```

written by yuxiangmusic original link here

Solution 3

```
vector<int> findPermutation(string s) {
  vector<int> ret;
  for (int i = 0; i <= s.size(); ++i)
    if (i == s.size() || s[i] == 'I')
      for (int j = i + 1, lenTmp = ret.size(); j > lenTmp; --j)
        ret.push_back(j);
  return ret;
}
```

written by lee215 original link here

From Leetcoder.

Max Consecutive Ones

Given a binary array, find the maximum number of consecutive 1s in this array.

Example 1:

```
Input: [1,1,0,1,1,1]
Output: 3
Explanation: The first two digits or the last three digits are consecutive 1s.
    The maximum number of consecutive 1s is 3.
```

Note:

- The input array will only contain `0` and `1`.
- The length of input array is a positive integer and will not exceed 10,000

Solution 1

```java
    public int findMaxConsecutiveOnes(int[] nums) {
        int maxHere = 0, max = 0;
        for (int n : nums)
            max = Math.max(max, maxHere = n == 0 ? 0 : maxHere + 1);
        return max;
    }
```

The idea is to reset `maxHere` to 0 if we see 0, otherwise increase `maxHere` by 1
The max of all `maxHere` is the solution

```
110111
^ maxHere = 1

110111
.^ maxHere = 2

110111
..^ maxHere = 0

110111
...^ maxHere = 1

110111
....^ maxHere = 2

110111
.....^ maxHere = 3
```

We can also solve this problem by setting `k = 0` of Max Consecutive Ones II

written by yuxiangmusic original link here

Solution 2

This is a really easy problem. No explanation :)

```
public class Solution {
    public int findMaxConsecutiveOnes(int[] nums) {
        int result = 0;
        int count = 0;

        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == 1) {
         count++;
         result = Math.max(count, result);
            }
            else count = 0;
        }

        return result;
    }
}
```

written by shawngao original link here

Solution 3

```
class Solution(object):
    def findMaxConsecutiveOnes(self, nums):
        cnt = 0
        ans = 0
        for num in nums:
            if num == 1:
                cnt += 1
                ans = max(ans, cnt)
            else:
                cnt = 0
        return ans
```

written by Ipeq1 original link here

From Leetcoder.

Predict the Winner

Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by the player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player. This continues until all the scores have been chosen. The player with the maximum score wins.

Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.

Example 1:

```
Input: [1, 5, 2]
Output: False
Explanation: Initially, player 1 can choose between 1 and 2.
If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If player 2 c
hooses 5, then player 1 will be left with 1 (or 2).
So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.
Hence, player 1 will never be the winner and you need to return False.
```

Example 2:

```
Input: [1, 5, 233, 7]
Output: True
Explanation: Player 1 first chooses 1. Then player 2 have to choose between 5 and 7
. No matter which number player 2 choose, player 1 can choose 233.
Finally, player 1 has more score (234) than player 2 (12), so you need to return Tr
ue representing player1 can win.
```

Note:

1. 1
2. Any scores in the given array are non-negative integers and will not exceed 10,000,000.
3. If the scores of both players are equal, then player 1 is still the winner.

Solution 1

```
public class Solution {
    public boolean PredictTheWinner(int[] nums) {
        return helper(nums, 0, nums.length-1)>=0;
    }
    private int helper(int[] nums, int s, int e){
        return s==e ? nums[e] : Math.max(nums[e] - helper(nums, s, e-1), nums[s]
- helper(nums, s+1, e));
    }
}
```

Inspired by @sameer13, add a cache:

```
public class Solution {
    public boolean PredictTheWinner(int[] nums) {
        return helper(nums, 0, nums.length-1, new Integer[nums.length][nums.lengt
h])>=0;
    }
    private int helper(int[] nums, int s, int e, Integer[][] mem){
        if(mem[s][e]==null)
            mem[s][e] = s==e ? nums[e] : Math.max(nums[e]-helper(nums,s,e-1,mem),
nums[s]-helper(nums,s+1,e,mem));
        return mem[s][e];
    }
}
```

Explanation
So assuming the sum of the array it SUM, so eventually player1 and player2 will split the SUM between themselves. For player1 to win, he/she has to get more than what player2 gets. If we think from the prospective of one player, then what he/she gains each time is a **plus**, while, what the other player gains each time is a**minus**. Eventually if player1 can have a >0 total, player1 can win.

Helper function simulate this process. In each round:
if e==s, there is no choice but have to select nums[s]
otherwise, this current player has 2 options:
--> nums[s]-helper(nums,s+1,e): this player select the front item, leaving the other player a choice from s+1 to e
--> nums[e]-helper(nums,s,e-1): this player select the tail item, leaving the other player a choice from s to e-1
Then take the max of these two options as this player's selection, return it.

written by chidong original link here

Solution 2

The main idea is each player will play optimally,
so if there exist that my opponent will lose the game (false), I will return true, that's why I use '!' in the return;

For example : [1, 5, 2]

```
if (player1 pick 1) the rest is [5, 2];
 if (player2 pick 2) player1 win(true),
 if (player2 pick 5) player1 lose(false),
-> because player2 play optimally, so he choose to pick 5,
   player2 = !(pick 2) || !(pick 5)  = !true || !false = true;
   so when player1 first choose 1, he always loses.

if (player1 pick 2) the rest is [1, 5];
 if (player2 pick 1) player1 win(true),
 if (player2 pick 5) player1 lose(false),
-> because player2 play optimally, so he choose to pick 5,
   player2 = !(pick 1) || !(pick 5)  = !true || !false = true;
   so when player1 first choose 2, he always loses.

So, it this case, no matter player1 first choose 1 or 2, he always loses.
   player1 = !(pick 1) || !(pick 2)  = !true || !true = false;
```

Here is my code:

```
public boolean PredictTheWinner(int[] nums) {
 return first(0, 0, nums, 0, nums.length-1);
}

private boolean first(int s1, int s2, int[] nums, int start, int end) {
 if (start >= end ){
  if (s1 >= s2) return true;
  else return false;
 }
 return !second(s1+nums[start], s2, nums, start+1, end) || !second(s1+nums[end],
s2, nums, start, end-1);
}

private boolean second(int s1, int s2, int[] nums, int start, int end) {
 if (start >= end ){
  if (s1 < s2) return true;
  else return false;
 }
 return !first(s1, s2+nums[start], nums, start+1, end) || !first(s1, s2+nums[end]
, nums, start, end-1);
}
```

written by shawloatrchen original link here

Solution 3

The idea is that this is a minimax game, and if you went to MIT and took 6.046 then you would have seen something similar to this problem in class. And thanks to MIT OCW everyone can see the explanation

The DP solution

```
class Solution {
public:
    bool PredictTheWinner(vector<int>& nums) {
        if(nums.size()% 2 == 0) return true;

        int n = nums.size();
        vector<vector<int>> dp(n, vector<int>(n, -1));

        int myBest = utill(nums, dp, 0, n-1);
        return 2*myBest >= accumulate(nums.begin(), nums.end(), 0);
    }

    int utill(vector<int>& v, vector<vector<int>> &dp, int i, int j){
        if(i > j) return 0;
        if(dp[i][j] != -1) return dp[i][j];

        int a = v[i] + min(utill(v,dp, i+1, j-1), utill(v, dp, i+2, j));
        int b = v[j] + min(utill(v,dp,i, j-2), utill(v,dp, i+1, j-1));
        dp[i][j] = max(a, b);

        return dp[i][j];
    }
};
```

written by kevin36 original link here

From Leetcoder.

Max Consecutive Ones II

Given a binary array, find the maximum number of consecutive 1s in this array if you can flip at most one 0.

Example 1:

```
Input: [1,0,1,1,0]
Output: 4
Explanation: Flip the first zero will get the the maximum number of consecutive 1s.
    After flipping, the maximum number of consecutive 1s is 4.
```

Note:

- The input array will only contain `0` and `1`.
- The length of input array is a positive integer and will not exceed 10,000

Follow up:

What if the input numbers come in one by one as an **infinite stream**? In other words, you can't store all numbers coming from the stream as it's too large to hold in memory. Could you solve it efficiently?

Solution 1

The idea is to keep a window `[l, h]` that contains at most `k` zero

The following solution does not handle follow-up, because `nums[l]` will need to access previous input stream
`Time: O(n) Space: O(1)`

```
    public int findMaxConsecutiveOnes(int[] nums) {
        int max = 0, zero = 0, k = 1; // flip at most k zero
        for (int l = 0, h = 0; h < nums.length; h++) {
            if (nums[h] == 0)
                zero++;
            while (zero > k)
                if (nums[l++] == 0)
                    zero--;
            max = Math.max(max, h - l + 1);
        }
        return max;
    }
```

Now let's deal with follow-up, we need to store up to `k` indexes of zero within the window `[l, h]` so that we know where to move `l` next when the window contains more than `k` zero. If the input stream is infinite, then the output could be extremely large because there could be super long consecutive ones. In that case we can use `BigInteger` for all indexes. For simplicity, here we will use `int`
`Time: O(n) Space: O(k)`

```
    public int findMaxConsecutiveOnes(int[] nums) {
        int max = 0, k = 1; // flip at most k zero
        Queue<Integer> zeroIndex = new LinkedList<>();
        for (int l = 0, h = 0; h < nums.length; h++) {
            if (nums[h] == 0)
                zeroIndex.offer(h);
            if (zeroIndex.size() > k)
                l = zeroIndex.poll() + 1;
            max = Math.max(max, h - l + 1);
        }
        return max;
    }
```

Note that setting `k = 0` will give a solution to the earlier version Max Consecutive Ones

For `k = 1` we can apply the same idea to simplify the solution. Here `q` stores the index of zero within the window `[l, h]` so its role is similar to `Queue` in the above solution

```java
    public int findMaxConsecutiveOnes(int[] nums) {
        int max = 0, q = -1;
        for (int l = 0, h = 0; h < nums.length; h++) {
            if (nums[h] == 0) {
                l = q + 1;
                q = h;
            }
            max = Math.max(max, h - l + 1);
        }
        return max;
    }
```

written by yuxiangmusic original link here

Solution 2

The solution for the previous problem "Max Consecutive Ones" is as simple as this

```
class Solution(object):
    def findMaxConsecutiveOnes(self, nums):
        nums.append(0)
        lo = 0
        ret = 0
        for hi,n in enumerate(nums):
            if n==0:
                ret = max(ret, hi-lo)
                lo = hi+1
        return ret
```

Based on the above solution, for "Max Consecutive Ones II", we simply keep track of 2 low pointers:

```
class Solution(object):
    def findMaxConsecutiveOnes(self, nums):
        nums.append(0)
        lo1, lo2 = 0, 0
        ret = 0
        for hi,n in enumerate(nums):
            if n==0:
                ret = max(ret, hi-lo1)
                lo1, lo2 = lo2, hi+1
        return ret
```

A code golfing version just for fun:

```
class Solution(object):
    def findMaxConsecutiveOnes(self, nums):
        return reduce(lambda l,e:(max(l[0],e[0]-l[1]),[l[2],l[1]][e[1]>0],[e[0]+1
,l[2]][e[1]>0]),enumerate(nums+[0]),[0,0,0])[0]
```

written by o_sharp original link here

Solution 3

```
public int findMaxConsecutiveOnes(int[] nums) {
     int maxConsecutive = 0, zeroLeft = 0, zeroRight = 0;
     for (int i=0;i<nums.length;i++) {
         zeroRight++;
         if (nums[i] == 0) {
             zeroLeft = zeroRight;
             zeroRight = 0;
         }
         maxConsecutive = Math.max(maxConsecutive, zeroLeft+zeroRight);
     }
     return maxConsecutive;
}
```

written by compton_scatter original link here

From Leetcoder.

Zuma Game

Think about Zuma Game. You have a row of balls on the table, colored red(R), yellow(Y), blue(B), green(G), and white(W). You also have several balls in your hand.

Each time, you may choose a ball in your hand, and insert it into the row (including the leftmost place and rightmost place). Then, if there is a group of 3 or more balls in the same color touching, remove these balls. Keep doing this until no more balls can be removed.

Find the minimal balls you have to insert to remove all the balls on the table. If you cannot remove all the balls, output -1.

```
Examples:

Input: "WRRBBW", "RB"
Output: -1
Explanation: WRRBBW -> WRR[R]BBW -> WBBW -> WBB[B]W -> WW

Input: "WWRRBBWW", "WRBRW"
Output: 2
Explanation: WWRRBBWW -> WWRR[R]BBWW -> WWBBWW -> WWBB[B]WW -> WWWW -> empty

Input:"G", "GGGGG"
Output: 2
Explanation: G -> G[G] -> GG[G] -> empty

Input: "RBYYBBRRB", "YRBGB"
Output: 3
Explanation: RBYYBBRRB -> RBYY[Y]BBRRB -> RBBBRRB -> RRRB -> B -> B[B] -> BB[B] ->
empty
```

Note:

1. You may assume that the initial row of balls on the table won't have any 3 or more consecutive balls with the same color.
2. The number of balls on the table won't exceed 20, and the string represents these balls is called "board" in the input.
3. The number of balls in your hand won't exceed 5, and the string represents these balls is called "hand" in the input.
4. Both input strings will be non-empty and only contain characters 'R','Y','B','G','W'.

Solution 1

```java
public class Solution {
    public int findMinStep(String board, String hand) {
        List<Character> boardList = new ArrayList<Character>();
        for (char c : board.toCharArray()) {
            boardList.add(c);
        }
        Map<Character,Integer> handMap = new HashMap<>();
        handMap.put('R',0);
        handMap.put('Y',0);
        handMap.put('B',0);
        handMap.put('G',0);
        handMap.put('W',0);
        for (char h : hand.toCharArray()) {
            handMap.put(h, handMap.get(h) + 1);
        }
        return find(boardList, handMap);
    }

    private int find(List<Character> board, Map<Character, Integer> hand) {
        cleanupBoard(board);
        if (board.size() == 0) return 0;
        if (empty(hand)) return -1;
        int count = 0;
        int min = Integer.MAX_VALUE;
        for (int i = 0; i<board.size(); i++) {
            char c = board.get(i);
            count++;
            if (i == board.size() - 1 || board.get(i+1) != c) {
                int missing = 3 - count;
                if (hand.get(c) >= missing) {
                    hand.put(c, hand.get(c) - missing);
                    List<Character> smallerBoard = new ArrayList<>(board);
                    for (int j = 0; j<count; j++) {
                        smallerBoard.remove(i-j);
                    }
                    int smallerFind = find(smallerBoard, hand);
                    if ( smallerFind != -1 ) {
                        min = Math.min(smallerFind + missing, min);
                    }
                    hand.put(c, hand.get(c) + missing);
                }
                count = 0;
            }
        }
        return (min == Integer.MAX_VALUE) ? -1 : min;
    }

    private void cleanupBoard(List<Character> board) {
        int count = 0;
        boolean cleaned = false;
        for (int i = 0; i<board.size(); i++) {
            char c = board.get(i);
            count++;
            if (i == board.size() - 1 || board.get(i+1) != c) {
```

```
                if (count >= 3) {
                    for (int j = 0; j<count; j++) {
                        board.remove(i-j);
                    }
                    cleaned = true;
                    break;
                }
                count = 0;
            }
        }
        if (cleaned) {
            cleanupBoard(board);
        }
    }

    private boolean empty(Map<Character,Integer> hand) {
        for (int val : hand.values()) {
            if (val > 0) return false;
        }
        return true;
    }
}
```

written by ruben3 original link here

Solution 2

At the first look, it is similar to Can I Win. But not like that problem, the `board` state is not relevant to the state of `hand` and this problem requires the minimum value. So, it seems that memorization might not help much. So, I just do it brute-force and stop iff it is sure there no better results than current one.

Sort `hand` string so that we easily know if there at least 2 balls with same color in hand to eliminate a single ball on board. Refer to inline comments for details.

According to the description, the number of balls in hand won't exceed `5`, to make life easier, I just return a number equal or greater than `6` when there is no way to clear board.

```
#define MAX_STEP 6
class Solution {
public:
    int findMinStep(string board, string hand) {
        sort(hand.begin(), hand.end());
        int res = helper(board, hand);
        return res > hand.size() ? -1 : res;
    }

    int helper(string b, string h) {
        if (b.empty()) return 0;
        if (h.empty()) return MAX_STEP;
        int res = MAX_STEP;
        for (int i = 0; i < h.size(); i++) {
            int j = 0;
            int n = b.size();
            while (j < n) {
                int k = b.find(h[i], j);
                if (k == string::npos) break;
                if (k < n-1 && b[k] == b[k+1]) { // 2 consecutive balls with same
color on board
                    string nextb = shrink(b.substr(0, k) + b.substr(k+2)); // shr
ink the string until no 3 or more consecutive balls in same color
                    if (nextb.empty()) return 1; // this is the best result for c
urrent board, no need to continue
                    string nexth = h;
                    nexth.erase(i, 1); // remove the used ball from hand
                    res = min(res, helper(nextb, nexth) + 1);
                    k++;
                }
                else if (i > 0 && h[i] == h[i-1]) { // 2 balls with same color in
hand
                    string nextb = shrink(b.substr(0, k) + b.substr(k+1)); // shr
ink the string until no 3 or more consecutive balls in same color
                    if (nextb.empty()) return 2;  // this is the best result for
current board, no need to continue
                    string nexth = h;
                    nexth.erase(i, 1); // remove the used balls from hand
                    nexth.erase(i-1, 1);
                    res = min(res, helper(nextb, nexth) + 2);
                }
```

```
                j = k + 1;
            }
        }
        return res;
    }

    string shrink(string s) {
        while(s.size() > 0) {
            int start = 0;
            bool done = true;
            for (int i = 0; i <= s.size(); i++) {
                if (i == s.size() || s[i] != s[start]) {
                    if (i - start >= 3) {
                        s = s.substr(0, start) + s.substr(i);
                        done = false;
                        break;
                    }
                    start = i;
                }
            }
            if (done) break;
        }
        return s;
    }
};
```

written by Hcisly original link here

Solution 3

Just searching and memorizing...
It is not a good interview question because many codes are not related to the algorithm.

```
public class Solution {
     public int findMinStep(String board, String hand) {
         HashMap<Character, int[]> mhand = new HashMap<Character, int[]>();
         mhand.put('R', new int[]{0});
         mhand.put('Y', new int[]{0});
         mhand.put('B', new int[]{0});
         mhand.put('G', new int[]{0});
         mhand.put('W', new int[]{0});
         for(char c:hand.toCharArray()){
                 mhand.get(c)[0]++;
         }
         HashMap<String, Integer> record = new HashMap<String, Integer>();
         int min = helper(board, mhand, record);
         if(min>=10000){
             return -1;
         } else {
             return min;
         }
     }

     int helper(String board, HashMap<Character, int[]> hand, HashMap<String, Int
eger> record){
         if(board.length()==0){
             return 0;
         } else {
             int min = 10000;
                for(int i=0;i<board.length();i++){
                    if(hand.get(board.charAt(i))[0]>0){
                        hand.get(board.charAt(i))[0]--;
                        String newboard = board.substring(0,i)+board.charAt(i)+bo
ard.substring(i);
                        newboard = further(newboard);
                        String c = code(newboard, hand);
                     if(record.containsKey(c)){
                      min = Math.min(min, 1+record.get(c));
                     } else {
                      int s = helper(newboard, hand, record);
                      min = Math.min(min, 1+s);
                      record.put(c, s);
                     }
                        hand.get(board.charAt(i))[0]++;
                    }
                }
             return min;
         }
     }
     String further(String board){
         if(board.length()==0){
             return "";
         }
```

```
        int count=1;
        for(int i=1;i<board.length();i++){
            if(board.charAt(i-1)==board.charAt(i)){
                count++;
            } else {
                if(count>=3){
                    return further(board.substring(0, i-count)+board.substring(i
));
                } else {
                    count=1;
                }
            }
        }
        if(count>=3){
            return board.substring(0, board.length()-count);
        }
        return board;
    }
    String code(String board, HashMap<Character, int[]> hand){
     StringBuilder sb = new StringBuilder();
     sb.append(board);
     for(Map.Entry<Character, int[]> e: hand.entrySet()){
      sb.append(e.getKey());
      sb.append(e.getValue()[0]);
     }
     return sb.toString();
    }
}
```

written by pureklkl original link here

From Leetcoder.

Increasing Subsequences

Given an integer array, your task is to find all the different possible increasing subsequences of the given array, and the length of an increasing subsequence should be at least 2 .

Example:

```
Input: [4, 6, 7, 7]
Output: [[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7,7], [4,7,7]
]
```

Note:

1. The length of the given array will not exceed 15.
2. The range of integer in the given array is [-100,100].
3. The given array may contain duplicates, and two equal integers should also be considered as a special case of increasing sequence.

Solution 1

```
public class Solution {

     public List<List<Integer>> findSubsequences(int[] nums) {
         Set<List<Integer>> res= new HashSet<List<Integer>>();
         List<Integer> holder = new ArrayList<Integer>();
         findSequence(res, holder, 0, nums);
         List result = new ArrayList(res);
         return result;
     }

    public void findSequence(Set<List<Integer>> res, List<Integer> holder, int in
dex, int[] nums) {
        if (holder.size() >= 2) {
            res.add(new ArrayList(holder));
        }
        for (int i = index; i < nums.length; i++) {
            if(holder.size() == 0 || holder.get(holder.size() - 1) <= nums[i]) {
                holder.add(nums[i]);
                findSequence(res, holder, i + 1, nums);
                holder.remove(holder.size() - 1);
            }
        }
    }
}
```

written by adarshkp original link here

Solution 2

```
public List<List<Integer>> findSubsequences(int[] nums) {
 List<List<Integer>> res = new ArrayList<>();
 helper(res, new ArrayList<Integer>(), nums, 0);
 return res;
}

private void helper(List<List<Integer>> res, List<Integer> list, int[] nums, int
id) {
 if (list.size() > 1) res.add(new ArrayList<Integer>(list));
 List<Integer> unique = new ArrayList<Integer>();
 for (int i = id; i < nums.length; i++) {
  if (id > 0 && nums[i] < nums[id-1]) continue; // skip non-increase
  if (unique.contains(nums[i])) continue; // skip duplicate
  unique.add(nums[i]);
  list.add(nums[i]);
  helper(res, list, nums, i+1);
  list.remove(list.size()-1);
 }
}
```

written by shawloatrchen original link here

Solution 3

```
class Solution {
private:
    void findSubsequences(vector<int> &nums, vector<int> &subsequence,
                          set<vector<int> > &result, int size, int index) {
    if (size >= 2)
        result.insert(subsequence);


    for (int i = index; i < nums.size(); ++i) {
        if (subsequence.size() == 0 || nums[i] >= subsequence[subsequence.size()
- 1]) {
            subsequence.push_back(nums[i]);
            findSubsequences(nums, subsequence, result, size + 1, i + 1);
            subsequence.pop_back();
        }
    }
}

public:
    vector<vector<int>> findSubsequences(vector<int>& nums) {
        set<vector<int>> resultSet;
        vector<int> subsequence;
        findSubsequences(nums, subsequence, resultSet, 0, 0);

        vector<vector<int>> result(resultSet.begin(), resultSet.end());
        return result;
    }
};
```

written by pratheek189 original link here

From Leetcoder.

Construct the Rectangle

For a web developer, it is very important to know how to design a web page's size. So, given a specific rectangular web page’s area, your job by now is to design a rectangular web page, whose length L and width W satisfy the following requirements:

```
1. The area of the rectangular web page you designed must equal to the given target
 area.

2. The width W should not be larger than the length L, which means L >= W.

3. The difference between length L and width W should be as small as possible.
```

You need to output the length L and the width W of the web page you designed in sequence.

Example:

```
Input: 4
Output: [2, 2]
Explanation: The target area is 4, and all the possible ways to construct it are [1
,4], [2,2], [4,1].
But according to requirement 2, [1,4] is illegal; according to requirement 3,  [4,1
] is not optimal compared to [2,2]. So the length L is 2, and the width W is 2.
```

Note:

1. The given area won't exceed 10,000,000 and is a positive integer
2. The web page's width and length you designed must be positive integers.

Solution 1

The W is always less than or equal to the square root of area
so we start searching at sqrt(area) till we find the result

```
public int[] constructRectangle(int area) {
        int w = (int)Math.sqrt(area);
 while (area%w!=0) w--;
 return new int[]{area/w, w};
}
```

written by shawloatrchen original link here

Solution 2

```
public int[] constructRectangle(int area) {
        int[] result = new int[2];
        if(area == 0){
            return result;
        }
        int a = (int)Math.sqrt(area);
        while(area%a != 0){
            a--;
        }
        int b = area/a;
        result[0] = b;
        result[1] = a;
        return result;
    }
```

written by prateek470 original link here

Solution 3

```
class Solution(object):
    def constructRectangle(self, area):
        mid = int(math.sqrt(area))
        while mid > 0:
            if area % mid == 0:
                return [int(area / mid), int(mid)]
            mid -= 1
```

written by Ipeq1 original link here

From Leetcoder.

Target Sum

You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols `+` and `-`. For each integer, you should choose one from `+` and `-` as its new symbol.

Find out how many ways to assign symbols to make sum of integers equal to target S.

Example 1:

```
Input: nums is [1, 1, 1, 1, 1], S is 3.
Output: 5
Explanation:

-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
+1+1+1-1+1 = 3
+1+1+1+1-1 = 3

There are 5 ways to assign symbols to make the sum of nums be target 3.
```

Note:

1. The length of the given array is positive and will not exceed 20.
2. The sum of elements in the given array will not exceed 1000.
3. Your output answer is guaranteed to be fitted in a 32-bit integer.

Solution 1

The recursive solution is very slow, because its runtime is exponential

The original problem statement is equivalent to:
Find a **subset** of `nums` that need to be positive, and the rest of them negative, such that the sum is equal to `target`

Let `P` be the positive subset and `N` be the negative subset
For example:
Given `nums = [1, 2, 3, 4, 5]` and `target = 3` then one possible solution is `+1-2+3-4+5 = 3`
Here positive subset is `P = [1, 3, 5]` and negative subset is `N = [2, 4]`

Then let's see how this can be converted to a subset sum problem:

```
                    sum(P) - sum(N) = target
sum(P) + sum(N) + sum(P) - sum(N) = target + sum(P) + sum(N)
                         2 * sum(P) = target + sum(nums)
```

So the original problem has been converted to a subset sum problem as follows:
Find a **subset** `P` of `nums` such that `sum(P) = (target + sum(nums)) / 2`

Note that the above formula has proved that `target + sum(nums)` must be even
We can use that fact to quickly identify inputs that do not have a solution (Thanks to @BrunoDeNadaiSarnaglia for the suggestion)
For detailed explanation on how to solve subset sum problem, you may refer to Partition Equal Subset Sum

Here is Java solution (15 ms)

```
    public int findTargetSumWays(int[] nums, int s) {
        int sum = 0;
        for (int n : nums)
            sum += n;
        return sum < s || (s + sum) % 2 > 0 ? 0 : subsetSum(nums, (s + sum) >>> 1
);
    }

    public int subsetSum(int[] nums, int s) {
        int[] dp = new int[s + 1];
        dp[0] = 1;
        for (int n : nums)
            for (int i = s; i >= n; i--)
                dp[i] += dp[i - n];
        return dp[s];
    }
```

Here is C++ solution (3 ms)

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int s) {
        int sum = accumulate(nums.begin(), nums.end(), 0);
        return sum < s || (s + sum) & 1 ? 0 : subsetSum(nums, (s + sum) >> 1);
    }

    int subsetSum(vector<int>& nums, int s) {
        int dp[s + 1] = { 0 };
        dp[0] = 1;
        for (int n : nums)
            for (int i = s; i >= n; i--)
                dp[i] += dp[i - n];
        return dp[s];
    }
};
```

written by yuxiangmusic original link here

Solution 2

```
public class Solution {
    public int findTargetSumWays(int[] nums, int s) {
        int sum = 0;
        for(int i: nums) sum+=i;
        if(s>sum || s<-sum) return 0;
        int[] dp = new int[2*sum+1];
        dp[0+sum] = 1;
        for(int i = 0; i<nums.length; i++){
            int[] next = new int[2*sum+1];
            for(int k = 0; k<2*sum+1; k++){
                if(dp[k]!=0){
                    next[k + nums[i]] += dp[k];
                    next[k - nums[i]] += dp[k];
                }
            }
            dp = next;
        }
        return dp[sum+s];
    }
}
```

Please up vote if it helped, so that more people can see it. ;)

written by Chidong original link here

Solution 3

I'm quite surprised that simple DFS could pass the test since for DFS solution there are obvious a lot of overlap subproblems. So I used a map to record the intermediate result while we are walking along the recursion tree.

```
public class Solution {
    public int findTargetSumWays(int[] nums, int S) {
        if (nums == null || nums.length == 0){
            return 0;
        }
        return helper(nums, 0, 0, S, new HashMap<>());
    }
    private int helper(int[] nums, int index, int sum, int S, Map<String, Integer
> map){
        String encodeString = index + "->" + sum;
        if (map.containsKey(encodeString)){
            return map.get(encodeString);
        }
        if (index == nums.length){
            if (sum == S){
                return 1;
            }else {
                return 0;
            }
        }
        int curNum = nums[index];
        int add = helper(nums, index + 1, sum - curNum, S, map);
        int minus = helper(nums, index + 1, sum + curNum, S, map);
        map.put(encodeString, add + minus);
        return add + minus;
    }
}
```

written by butzhang original link here

From Leetcoder.

Teemo Attacking

In LLP world, there is a hero called Teemo and his attacking can make his enemy Ashe be in poisoned condition. Now, given the Teemo's attacking **ascending** time series towards Ashe and the poisoning time duration per Teemo's attacking, you need to output the total time that Ashe is in poisoned condition.

You may assume that Teemo attacks at the very beginning of a specific time point, and makes Ashe be in poisoned condition immediately.

Example 1:

```
Input: [1,4], 2
Output: 4
Explanation: At time point 1, Teemo starts attacking Ashe and makes Ashe be poisone
d immediately.
This poisoned status will last 2 seconds until the end of time point 2.
And at time point 4, Teemo attacks Ashe again, and causes Ashe to be in poisoned st
atus for another 2 seconds.
So you finally need to output 4.
```

Example 2:

```
Input: [1,2], 2
Output: 3
Explanation: At time point 1, Teemo starts attacking Ashe and makes Ashe be poisone
d.
This poisoned status will last 2 seconds until the end of time point 2.
However, at the beginning of time point 2, Teemo attacks Ashe again who is already
in poisoned status.
Since the poisoned status won't add up together, though the second poisoning attack
 will still work at time point 2, it will stop at the end of time point 3.
So you finally need to output 3.
```

Note:

1. You may assume the length of given time series array won't exceed 10000.
2. You may assume the numbers in the Teemo's attacking time series and his poisoning time duration per attacking are non-negative integers, which won't exceed 10,000,000.

Solution 1

```
class Solution(object):
    def findPoisonedDuration(self, timeSeries, duration):
        ans = duration * len(timeSeries)
        for i in range(1,len(timeSeries)):
            ans -= max(0, duration - (timeSeries[i] - timeSeries[i-1]))
        return ans
```

written by lilixu original link here

Solution 2

For each `begin` followed by `t`
If `t` is within previous duration `[begin, begin + duration]` then increase total by `t - begin`
If `t` in out of previous duration `[begin, begin + duration]` then increase total by `duration`
In both cases update `begin` to the new begin time `t`

```
public int findPoisonedDuration(int[] timeSeries, int duration) {
    if (timeSeries.length == 0) return 0;
    int begin = timeSeries[0], total = 0;
    for (int t : timeSeries) {
        total+= t < begin + duration ? t - begin : duration;
        begin = t;
    }
    return total + duration;
}
```

written by yuxiangmusic original link here

Solution 3

The same idea as https://leetcode.com/problems/merge-intervals/
Algorithm:

1. Use two variable to record current start and end point.
2. If the start of new interval is greater than current end, meaning NO overlapping, we can sum the current interval length to result and then update start and end.
3. Otherwise just update the current end;

```
public class Solution {
    public int findPosisonedDuration(int[] timeSeries, int duration) {
        if (timeSeries == null || timeSeries.length == 0 || duration == 0) return
0;

        int result = 0, start = timeSeries[0], end = timeSeries[0] + duration;
        for (int i = 1; i < timeSeries.length; i++) {
            if (timeSeries[i] > end) {
                result += end - start;
                start = timeSeries[i];
            }
            end = timeSeries[i] + duration;
        }
        result += end - start;

        return result;
    }
}
```

written by shawngao original link here

From Leetcoder.

Next Greater Element I

You are given two arrays **(without duplicates)** `nums1` and `nums2` where `nums1`'s elements are subset of `nums2`. Find all the next greater numbers for `nums1`'s elements in the corresponding places of `nums2`.

The Next Greater Number of a number **x** in `nums1` is the first greater number to its right in `nums2`. If it does not exist, output -1 for this number.

Example 1:

```
Input: nums1 = [4,1,2], nums2 = [1,3,4,2].
Output: [-1,3,-1]
Explanation:
    For number 4 in the first array, you cannot find the next greater number for it
 in the second array, so output -1.
    For number 1 in the first array, the next greater number for it in the second a
rray is 3.
    For number 2 in the first array, there is no next greater number for it in the
second array, so output -1.
```

Example 2:

```
Input: nums1 = [2,4], nums2 = [1,2,3,4].
Output: [3,-1]
Explanation:
    For number 2 in the first array, the next greater number for it in the second a
rray is 3.
    For number 4 in the first array, there is no next greater number for it in the
second array, so output -1.
```

Note:

1. All elements in `nums1` and `nums2` are unique.
2. The length of both `nums1` and `nums2` would not exceed 1000.

Solution 1

Key observation:
Suppose we have a decreasing sequence followed by a greater number
For example `[5, 4, 3, 2, 1, 6]` then the greater number `6` is the next greater element for all previous numbers in the sequence

We use a stack to keep a **decreasing** sub-sequence, whenever we see a number `x` greater than `stack.peek()` we pop all elements less than `x` and for all the popped ones, their next greater element is `x`
For example `[9, 8, 7, 3, 2, 1, 6]`
The stack will first contain `[9, 8, 7, 3, 2, 1]` and then we see `6` which is greater than `1` so we pop `1 2 3` whose next greater element should be `6`

```
    public int[] nextGreaterElement(int[] findNums, int[] nums) {
        Map<Integer, Integer> map = new HashMap<>(); // map from x to next greate
r element of x
        Stack<Integer> stack = new Stack<>();
        for (int num : nums) {
            while (!stack.isEmpty() && stack.peek() < num)
                map.put(stack.pop(), num);
            stack.push(num);
        }
        for (int i = 0; i < findNums.length; i++)
            findNums[i] = map.getOrDefault(findNums[i], -1);
        return findNums;
    }
```

written by yuxiangmusic original link here

Solution 2

```java
public class Solution {
    public int[] nextGreaterElement(int[] findNums, int[] nums) {
        int[] ret = new int[findNums.length];
        ArrayDeque<Integer> stack = new ArrayDeque<>();
        HashMap<Integer, Integer> map = new HashMap<>();
        for(int i = nums.length - 1; i >= 0; i--) {
            while(!stack.isEmpty() && stack.peek() <= nums[i]) {
                stack.pop();
            }
            if(stack.isEmpty()) map.put(nums[i], -1);
            else map.put(nums[i], stack.peek());
            stack.push(nums[i]);
        }
        for(int i = 0; i < findNums.length; i++) {
            ret[i] = map.get(findNums[i]);
        }
        return ret;
    }
}
```

written by zhengyuyu123 original link here

Solution 3

```
def nextGreaterElement(self, findNums, nums):
    return [next((y for y in nums[nums.index(x):] if y > x), -1) for x in findNum
s]
```

written by StefanPochmann original link here

From Leetcoder.

Diagonal Traverse

Given a matrix of M x N elements (M rows, N columns), return all elements of the matrix in diagonal order as shown in the below image.

Example:

```
Input:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
Output:  [1,2,4,7,5,3,6,8,9]
Explanation:
```

Note:

1. The total number of elements of the given matrix will not exceed 10,000.

Solution 1

I don't think this is a hard problem. It is easy to figure out the walk pattern. Anyway...
Walk patterns:

- If out of `bottom border` (row >= m) then row = m - 1; col += 2; change walk direction.
- if out of `right border` (col >= n) then col = n - 1; row += 2; change walk direction.
- if out of `top border` (row < 0) then row = 0; change walk direction.
- if out of `left border` (col < 0) then col = 0; change walk direction.
- Otherwise, just go along with the current direction.

Time complexity: O(m * n), m = number of rows, n = number of columns.
Space complexity: O(1).

```
public class Solution {
    public int[] findDiagonalOrder(int[][] matrix) {
        if (matrix == null || matrix.length == 0) return new int[0];
        int m = matrix.length, n = matrix[0].length;

        int[] result = new int[m * n];
        int row = 0, col = 0, d = 0;
        int[][] dirs = {{-1, 1}, {1, -1}};

        for (int i = 0; i < m * n; i++) {
            result[i] = matrix[row][col];
            row += dirs[d][0];
            col += dirs[d][1];

            if (row >= m) { row = m - 1; col += 2; d = 1 - d;}
            if (col >= n) { col = n - 1; row += 2; d = 1 - d;}
            if (row < 0)  { row = 0; d = 1 - d;}
            if (col < 0)  { col = 0; d = 1 - d;}
        }

        return result;
    }
}
```

written by shawngao original link here

Solution 2

Put all diagonal sequences from top-right to bottom-left to an array and then combine all sequence together by reversing odd sequences.

```
class Solution {
public:
    vector<int> findDiagonalOrder(vector<vector<int>>& matrix) {
        int m = matrix.size();
        if (m == 0) return vector<int>();
        int n = matrix[0].size();
        vector<vector<int>> tmp (m+n-1);
        for (int i = 0; i < m+n-1 ; i++) {
            int row = max(0, i-n+1);
            int col = min(i, n-1);
            for (; col >= 0 && row < m; row++, col--) {
                tmp[i].push_back(matrix[row][col]);
            }
        }
        vector<int> res;
        for (int i = 0; i< tmp.size(); i++) {
            if (i % 2) res.insert(res.end(), tmp[i].begin(), tmp[i].end());
            else res.insert(res.end(), tmp[i].rbegin(), tmp[i].rend());
        }
        return res;
    }
};
```

written by Hcisly original link here

Solution 3

```
public int[] findDiagonalOrder(int[][] matrix) {
 if (matrix.length == 0) return new int[0];
 int h = matrix.length, w = matrix[0].length, id = 0;
 int[] res  = new int[h*w];
 for (int i = 0; i < h+w; i++) {
  // find lower bound and upper bound
  int lb = (int)Math.max(0, i-w+1), ub = (int)Math.min(i,h-1);
  if (i%2 == 0) for (int j = ub; j >= lb; j--) res[id++] = matrix[j][i-j];
  else for (int j = lb;j <= ub; j++) res[id++] = matrix[j][i-j];
 }
 return res;
}
```

written by shawloatrchen original link here

From Leetcoder.

The Maze III

There is a **ball** in a maze with empty spaces and walls. The ball can go through empty spaces by rolling **up** (u), **down** (d), **left** (l) or **right** (r), but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction. There is also a **hole** in this maze. The ball will drop into the hole if it rolls on to the hole.

Given the **ball position**, the **hole position** and the **maze**, find out how the ball could drop into the hole by moving the **shortest distance**. The distance is defined by the number of **empty spaces** traveled by the ball from the start position (excluded) to the hole (included). Output the moving **directions** by using 'u', 'd', 'l' and 'r'. Since there could be several different shortest ways, you should output the **lexicographically smallest** way. If the ball cannot reach the hole, output "impossible".

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The ball and the hole coordinates are represented by row and column indexes.

Example 1

```
Input 1: a maze represented by a 2D array

0 0 0 0 0
1 1 0 0 1
0 0 0 0 0
0 1 0 0 1
0 1 0 0 0

Input 2: ball coordinate (rowBall, colBall) = (4, 3)
Input 3: hole coordinate (rowHole, colHole) = (0, 1)

Output: "lul"
Explanation: There are two shortest ways for the ball to drop into the hole.
The first way is left -> up -> left, represented by "lul".
The second way is up -> left, represented by 'ul'.
Both ways have shortest distance 6, but the first way is lexicographically smaller
because 'l'
```

Example 2

```
Input 1: a maze represented by a 2D array

0 0 0 0 0
1 1 0 0 1
0 0 0 0 0
0 1 0 0 1
0 1 0 0 0

Input 2: ball coordinate (rowBall, colBall) = (4, 3)
Input 3: hole coordinate (rowHole, colHole) = (3, 0)
Output: "impossible"
Explanation: The ball cannot reach the hole.
```

Note:

1. There is only one ball and one hole in the maze.
2. Both the ball and hole exist on an empty space, and they will not be at the same position initially.
3. The given maze does not contain border (like the red rectangle in the example pictures), but you could assume the border of the maze are all walls.
4. The maze contains at least 2 empty spaces, and the width and the height of the maze won't exceed 30.

Solution 1

```
public class Solution {
    int min; //min distance to hole
    String minS; //min distance's path string
    int[] hole;
    int[][] maze;
    int[][] map; //shortest distant traveling from ball to this point
    int[][] dirs = {{0,1},{-1,0},{1,0},{0,-1}}; //r, u, d, l
    public String findShortestWay(int[][] maze, int[] ball, int[] hole) {
        this.min = Integer.MAX_VALUE;
        this.minS = null;
        this.hole = hole;
        this.maze = maze;
        this.map = new int[maze.length][maze[0].length];
        for(int i = 0; i<map.length; i++) Arrays.fill(map[i], Integer.MAX_VALUE);


        move(ball[0], ball[1], 0, "", -1);
        return (minS==null) ? "impossible" : minS;
    }

    private void move(int r, int c, int cnt, String path, int dir){//dir is a inde
x of dirs
        if(cnt > min || cnt > map[r][c]) return; //not a shortest route for sure
        if(dir!=-1){//if not from start point
            //add path
            if(dir==0) path+='r';
            else if(dir==1) path+='u';
            else if(dir==2) path+='d';
            else path+='l';

            //roll along dir
            while(r>=0 && r<maze.length && c>=0 && c<maze[0].length && maze[r][c]
==0){
                map[r][c] = Math.min(map[r][c], cnt);
                if(r==hole[0] && c==hole[1]){//check hole
                    if(cnt==min && path.compareTo(minS)<0){
                        minS=path;
                    }else if(cnt<min){
                        min = cnt;
                        minS = path;
                    }
                    return;
                }
                r += dirs[dir][0];
                c += dirs[dir][1];
                cnt++;
            }
            r -= dirs[dir][0];//[r,c] is wall, need to walk back 1 step
            c -= dirs[dir][1];
            cnt--;
        }

        //hit wall (or start) -> try to turn
        for(int i = 0; i<dirs.length; i++){
```

```
            if(dir == i) continue;//dont keep going
            if(dir == (3-i)) continue;//dont go back
            int newR = r + dirs[i][0];
            int newC = c + dirs[i][1];
            if(newR>=0 && newR<maze.length && newC>=0 && newC<maze[0].length && m
aze[newR][newC]==0){//can go
                move(r, c, cnt, path, i);
            }
        }
    }
}
```

Each time, first add the direction to the path, and then go with that direction, checking for hole along the way. When hit a wall, try to turn, and go with the new direction. For the starting point, don't "go", jump directly to "turn" part.

written by Chidong original link here

Solution 2

We can use Dijkstra's algorithm to find the shortest distance from the ball to the hole. If you are unfamiliar with this algorithm, how it works is that we process events in priority order, where the priority is (distance, path_string). When an event is processed, it adds neighboring nodes with respective distance. To repeatedly find the highest priority node to process, we use a heap (priority queue or 'pq'), where we can add nodes with logarithmic time complexity, and maintains the invariant that pq[0] is always the smallest (highest priority.) When we reach the hole for the first time (if we do), we are guaranteed to have the right answer in terms of having the shortest distance and the lexicographically smallest path-string.

When we look for the neighbors of a location in the matrix, we simulate walking up/left/right/down as long as we are inside the bounds of the matrix and the path is clear. If during this simulation we reach the hole prematurely, we should also stop. If after searching with our algorithm it is the case that we never reached the hole, then the task is impossible.

```
def findShortestWay(self, A, ball, hole):
    ball, hole = tuple(ball), tuple(hole)
    R, C = len(A), len(A[0])

    def neighbors(r, c):
        for dr, dc, di in [(-1, 0, 'u'), (0, 1, 'r'),
                           (0, -1, 'l'), (1, 0, 'd')]:
            cr, cc, dist = r, c, 0
            while (0 <= cr + dr < R and
                    0 <= cc + dc < C and
                    not A[cr+dr][cc+dc]):
                cr += dr
                cc += dc
                dist += 1
                if (cr, cc) == hole:
                    break
            yield (cr, cc), di, dist

    pq = [(0, '', ball)]
    seen = set()
    while pq:
        dist, path, node = heapq.heappop(pq)
        if node in seen: continue
        if node == hole: return path
        seen.add(node)
        for nei, di, nei_dist in neighbors(*node):
            heapq.heappush(pq, (dist+nei_dist, path+di, nei) )

    return "impossible"
```

written by awice original link here

Solution 3

I am rolling the ball from the stopping point recursively in the down-left-right-up order, and keeping track of the path with the shortest distance. That way, the first discovered path will be lexicographically smallest (if distance is the same). Except the very first roll, the ball is rolled only vertically (down and up) or horizontally (left and right), alternating.

When the ball stops, I am using the existing maze vector to store the current distance. If the ball was there before, I only continue the search if the current distance is smaller.

```
string roll(vector<vector<int>>& maze, int rowBall, int colBall, const vector<int>
& hole,
    int d_row, int d_col, int steps, const string& path, pair<string, int>& res)
{
    if (steps < res.second) {
        if (d_row != 0 || d_col != 0) { // both are zero for the initial ball pos
ition.
            while ((rowBall + d_row) >= 0 && (colBall + d_col) >= 0 && (rowBall +
d_row) <  maze.size()
                && (colBall + d_col) < maze[0].size() && maze[rowBall + d_row][co
lBall + d_col] != 1)
            {
                rowBall += d_row;
                colBall += d_col;
                ++steps;
                if (rowBall == hole[0] && colBall == hole[1] && steps < res.secon
d) res = {path, steps};
            }
        }
        if (maze[rowBall][colBall] == 0 || steps + 2 < maze[rowBall][colBall]) {
            maze[rowBall][colBall] = steps + 2; // '1' is for the walls.
            if (d_row == 0) roll(maze, rowBall, colBall, hole, 1, 0, steps, path
+ "d", res);
            if (d_col == 0) roll(maze, rowBall, colBall, hole, 0, -1, steps, path
+ "l", res);
            if (d_col == 0) roll(maze, rowBall, colBall, hole, 0, 1, steps, path
+ "r", res);
            if (d_row == 0) roll(maze, rowBall, colBall, hole, -1, 0, steps, path
+ "u", res);
        }
    }
    return res.first;
}
string findShortestWay(vector<vector<int>>& maze, vector<int>& ball, vector<int>&
hole)
{
    return roll(maze, ball[0], ball[1], hole, 0, 0, 0, "", pair<string, int>() =
{"impossible", INT_MAX});
}
```

written by votrubac original link here

From Leetcode?

Keyboard Row

Given a List of words, return the words that can be typed using letters of**alphabet** on only one row's of American keyboard like the image below.

Example 1:

```
Input: ["Hello", "Alaska", "Dad", "Peace"]
Output: ["Alaska", "Dad"]
```

Note:

1. You may use one character in the keyboard more than once.
2. You may assume the input string will only contain letters of alphabet.

Solution 1

```
public String[] findWords(String[] words) {
    return Stream.of(words).filter(s -> s.toLowerCase().matches("[qwertyuiop]*|[a
sdfghjkl]*|[zxcvbnm]*")).toArray(String[]::new);
}
```

written by lixx2100 original link here

Solution 2

```
public class Solution {
    public String[] findWords(String[] words) {
        String[] strs = {"QWERTYUIOP","ASDFGHJKL","ZXCVBNM"};
        Map<Character, Integer> map = new HashMap<>();
        for(int i = 0; i<strs.length; i++){
            for(char c: strs[i].toCharArray()){
                map.put(c, i);//put <char, rowIndex> pair into the map
            }
        }
        List<String> res = new LinkedList<>();
        for(String w: words){
            if(w.equals("")) continue;
            int index = map.get(w.toUpperCase().charAt(0));
            for(char c: w.toUpperCase().toCharArray()){
                if(map.get(c)!=index){
                    index = -1; //don't need a boolean flag.
                    break;
                }
            }
            if(index!=-1) res.add(w);//if index != -1, this is a valid string
        }
        return res.toArray(new String[0]);
    }
}
```

written by Chidong original link here

Solution 3

Just regular expressions...

Ruby

```
def find_words(words)
  words.select { |w| w =~ /^([qwertyuiop]*|[asdfghjkl]*|[zxcvbnm]*)$/i }
end
```

Python

```
def findWords(self, words):
    return filter(re.compile('(?i)([qwertyuiop]*|[asdfghjkl]*|[zxcvbnm]*)$').matc
h, words)
```

written by StefanPochmann original link here

From Leetcoder.

Find Mode in Binary Search Tree

Given a binary search tree (BST) with duplicates, find all themode(s) (the most frequently occurred element) in the given BST.

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys**less than or equal to** the node's key.
- The right subtree of a node contains only nodes with keys**greater than or equal to** the node's key.
- Both the left and right subtrees must also be binary search trees.

For example:
Given BST `[1,null,2,2]`,

```
   1
    \
     2
    /
   2
```

return `[2]`.

Note: If a tree has more than one mode, you can return them in any order.

Follow up: Could you do that without using any extra space? (Assume that the implicit stack space incurred due to recursion does not count).

Solution 1

I've seen several solutions claimed to be O(1) space, but I disagree. They traverse the tree in in-order and keep track of the current set of modes (among other things). But that's not O(1) space, not even when disregarding recursion stack space (as explicitly allowed) and result space (not mentioned but reasonable). The set's contents aren't on stack space, so it can't be disregarded that way. And if the values are for example 1,2,3,4,...,n-2,n-1,n-1 (unique values followed by one double value), the set grows to Ω(n) and it can't be disregarded because the result only has size 1.

I think the way to do it properly is to do two passes. One to find the highest number of occurrences of any value, and then a second pass to collect all values occurring that often. Any other ideas?

Here's a (two-pass) solution that I think can rightfully be called O(1) space. Both passes keep track of the current value etc, and the second pass additionally collects the modes in the result array. I took the value handling out of the in-order traversal into its own function for clarity. Also, this way you could very easily replace my recursive in-order traversal with for example Morris traversal. Then you wouldn't even need to disregard the recursion stack space in order to claim O(1) extra space usage.

```java
public class Solution {

    public int[] findMode(TreeNode root) {
        inorder(root);
        modes = new int[modeCount];
        modeCount = 0;
        currCount = 0;
        inorder(root);
        return modes;
    }

    private int currVal;
    private int currCount = 0;
    private int maxCount = 0;
    private int modeCount = 0;

    private int[] modes;

    private void handleValue(int val) {
        if (val != currVal) {
            currVal = val;
            currCount = 0;
        }
        currCount++;
        if (currCount > maxCount) {
            maxCount = currCount;
            modeCount = 1;
        } else if (currCount == maxCount) {
            if (modes != null)
                modes[modeCount] = currVal;
            modeCount++;
        }
    }

    private void inorder(TreeNode root) {
        if (root == null) return;
        inorder(root.left);
        handleValue(root.val);
        inorder(root.right);
    }
}
```

Edit: Here's Morris traversal, just replace my previous `inorder` function with this. I hadn't realized it earlier, but having my separate `handleValue` function doesn't just nicely separate the traversal logic from this problem's logic, but it's also beneficial for Morris traversal because I'm calling `handleValue` from two different places in the code!

```java
    private void inorder(TreeNode root) {
        TreeNode node = root;
        while (node != null) {
            if (node.left == null) {
                handleValue(node.val);
                node = node.right;
            } else {
                TreeNode prev = node.left;
                while (prev.right != null && prev.right != node)
                    prev = prev.right;
                if (prev.right == null) {
                    prev.right = node;
                    node = node.left;
                } else {
                    prev.right = null;
                    handleValue(node.val);
                    node = node.right;
                }
            }
        }
    }
```

written by StefanPochmann original link here

Solution 2

What does "mode" mean?

written by ycc0602 original link here

Solution 3

O(n) time O(n) space

```
public class Solution {
    Map<Integer, Integer> map;
    int max = 0;
    public int[] findMode(TreeNode root) {
        if(root==null) return new int[0];
        this.map = new HashMap<>();

        inorder(root);

        List<Integer> list = new LinkedList<>();
        for(int key: map.keySet()){
            if(map.get(key) == max) list.add(key);
        }

        int[] res = new int[list.size()];
        for(int i = 0; i<res.length; i++) res[i] = list.get(i);
        return res;
    }

    private void inorder(TreeNode node){
        if(node.left!=null) inorder(node.left);
        map.put(node.val, map.getOrDefault(node.val, 0)+1);
        max = Math.max(max, map.get(node.val));
        if(node.right!=null) inorder(node.right);
    }
}
```

Just travel the tree and count, find the those with max counts. Nothing much. Spent 10min on figuring out what is mode....

If using this method (hashmap), inorder/preorder/postorder gives the same result. Because essentially you just travel the entire nodes and count. And BST is not necessary. This method works for any tree.

written by Chidong original link here

From Leetcoder.

IPO

Suppose LeetCode will start its IPO soon. In order to sell a good price of its shares to Venture Capital, LeetCode would like to work on some projects to increase its capital before the IPO. Since it has limited resources, it can only finish at most **k** distinct projects before the IPO. Help LeetCode design the best way to maximize its total capital after finishing at most **k** distinct projects.

You are given several projects. For each project **i**, it has a pure profit $\mathbf{P_i}$ and a minimum capital of $\mathbf{C_i}$ is needed to start the corresponding project. Initially, you have **W** capital. When you finish a project, you will obtain its pure profit and the profit will be added to your total capital.

To sum up, pick a list of at most **k** distinct projects from given projects to maximize your final capital, and output your final maximized capital.

Example 1:

```
Input: k=2, W=0, Profits=[1,2,3], Capital=[0,1,1].

Output: 4

Explanation: Since your initial capital is 0, you can only start the project indexe
d 0.
             After finishing it you will obtain profit 1 and your capital becomes 1
.
             With capital 1, you can either start the project indexed 1 or the proj
ect indexed 2.
             Since you can choose at most 2 projects, you need to finish the projec
t indexed 2 to get the maximum capital.
             Therefore, output the final maximized capital, which is 0 + 1 + 3 = 4.
```

Note:

1. You may assume all numbers in the input are non-negative integers.
2. The length of Profits array and Capital array will not exceed 50,000.
3. The answer is guaranteed to fit in a 32-bit signed integer.

Solution 1

The idea is each time we find a project with `max` profit and within current capital capability.
Algorithm:

1. Create (capital, profit) pairs and put them into PriorityQueue `pqCap`. This PriorityQueue sort by capital increasingly.
2. Keep polling pairs from `pqCap` until the project out of current capital capability. Put them into PriorityQueue `pqPro` which sort by profit decreasingly.
3. Poll one from `pqPro`, it's guaranteed to be the project with `max` profit and within current capital capability. Add the profit to capital `W`.
4. Repeat step 2 and 3 till finish `k` steps or no suitable project (pqPro.isEmpty()).

Time Complexity: For worst case, each project will be inserted and polled from both PriorityQueues once, so the overall runtime complexity should be `O(NlgN)`, N is number of projects.

```
public class Solution {
    public int findMaximizedCapital(int k, int W, int[] Profits, int[] Capital) {
        PriorityQueue<int[]> pqCap = new PriorityQueue<>((a, b) -> (a[0] - b[0]))
;
        PriorityQueue<int[]> pqPro  = new PriorityQueue<>((a, b) -> (b[1] - a[1])
);

        for (int i = 0; i < Profits.length; i++) {
            pqCap.add(new int[] {Capital[i], Profits[i]});
        }

        for (int i = 0; i < k; i++) {
            while (!pqCap.isEmpty() && pqCap.peek()[0] <= W) {
                pqPro.add(pqCap.poll());
            }

            if (pqPro.isEmpty()) break;

            W += pqPro.poll()[1];
        }

        return W;
    }
}
```

written by shawngao original link here

Solution 2

Loop k times:
Add all possible projects (Capital <= W) into the priority queue with the priority = -Profit.
Get the project with the smallest priority (biggest Profit).
Add the Profit to W

```
def findMaximizedCapital(self, k, W, Profits, Capital):
    import Queue
    q = Queue.PriorityQueue()
    projects = sorted(zip(Profits, Capital), key=lambda l: l[1])
    j = 0
    for i in range(k):
      while j < len(projects):
        if projects[j][1] > W:
          break
        else:
          q.put((-projects[j][0], projects[j][0]))
        j = j + 1
      if q.empty():
        break
      else:
        W += q.get()[1]
    return W
```

written by lee215 original link here

Solution 3

Keep a max-heap of current possible profits. Insert possible profits as soon as their needed capital is reached.

```
def findMaximizedCapital(self, k, W, Profits, Capital):
    current = []
    future = sorted(zip(Capital, Profits))[::-1]
    for _ in range(k):
        while future and future[-1][0] <= W:
            heapq.heappush(current, -future.pop()[1])
        if current:
            W -= heapq.heappop(current)
    return W
```

written by StefanPochmann original link here

From Leetcoder.

Next Greater Element II

Given a circular array (the next element of the last element is the first element of the array), print the Next Greater Number for every element. The Next Greater Number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, output -1 for this number.

Example 1:

```
Input: [1,2,1]
Output: [2,-1,2]
Explanation: The first 1's next greater number is 2; The number 2 can't find next g
reater number; The second 1's next greater number needs to search circularly, which
 is also 2.
```

Note: The length of given array won't exceed 10000.

Solution 1

The approach is same as *Next Greater Element I*
See explanation in my solution to the previous problem
The only difference here is that we use `stack` to keep the **indexes** of the **decreasing** subsequence

Java

```
    public int[] nextGreaterElements(int[] nums) {
        int n = nums.length, next[] = new int[n];
        Arrays.fill(next, -1);
        Stack<Integer> stack = new Stack<>(); // index stack
        for (int i = 0; i < n * 2; i++) {
            int num = nums[i % n];
            while (!stack.isEmpty() && nums[stack.peek()] < num)
                next[stack.pop()] = num;
            if (i < n) stack.push(i);
        }
        return next;
    }
```

C++

```
    vector<int> nextGreaterElements(vector<int>& nums) {
        int n = nums.size();
        vector<int> next(n, -1);
        stack<int> s; // index stack
        for (int i = 0; i < n * 2; i++) {
            int num = nums[i % n];
            while (!s.empty() && nums[s.top()] < num) {
                next[s.top()] = num;
                s.pop();
            }
            if (i < n) s.push(i);
        }
        return next;
    }
```

written by yuxiangmusic original link here

Solution 2

The first typical way to solve circular array problems is to extend the original array to twice length, 2nd half has the same element as first half. Then everything become simple.
Naive by simple solution, just look for the next greater element directly. Time complexity: O(n^2).

```
public class Solution {
    public int[] nextGreaterElements(int[] nums) {
        int max = Integer.MIN_VALUE;
        for (int num : nums) {
            max = Math.max(max, num);
        }

        int n = nums.length;
        int[] result = new int[n];
        int[] temp = new int[n * 2];

        for (int i = 0; i < n * 2; i++) {
            temp[i] = nums[i % n];
        }

        for (int i = 0; i < n; i++) {
            result[i] = -1;
            if (nums[i] == max) continue;

            for (int j = i + 1; j < n * 2; j++) {
                if (temp[j] > nums[i]) {
                    result[i] = temp[j];
                    break;
                }
            }
        }

        return result;
    }
}
```

The second way is to use a `stack` to facilitate the look up. First we put all indexes into the stack, `smaller` index on the `top`. Then we start from `end` of the array look for the first element (index) in the stack which is greater than the current one. That one is guaranteed to be the `Next Greater Element`. Then put the current element (index) into the stack.
Time complexity: O(n).

```java
public class Solution {
    public int[] nextGreaterElements(int[] nums) {
        int n = nums.length;
        int[] result = new int[n];

        Stack<Integer> stack = new Stack<>();
        for (int i = n - 1; i >= 0; i--) {
            stack.push(i);
        }

        for (int i = n - 1; i >= 0; i--) {
            result[i] = -1;
            while (!stack.isEmpty() && nums[stack.peek()] <= nums[i]) {
                stack.pop();
            }
            if (!stack.isEmpty()){
                result[i] = nums[stack.peek()];
            }
            stack.add(i);
        }

        return result;
    }
}
```

written by shawngao original link here

Solution 3

From viewing other solutions I think in some other languages O(n * n) are accepted. Can my code be improved for O(n * n) or do I have to use another language or approach?

```
class Solution(object):
    def nextGreaterElements(self, nums):

        if not nums:
            return []

        n = len(nums)
        result = [-1 for _ in range(len(nums))]
        max_num = max(nums)

        for i, num in enumerate(nums):

            if num == max_num:
                continue

            j = i
            while nums[j] <= num:
                j += 1
                j %= n
            result[i] = nums[j]

        return result
```

written by yorkshire original link here

From Leetcoder.

The Maze II

There is a **ball** in a maze with empty spaces and walls. The ball can go through empty spaces by rolling **up**, **down**, **left** or **right**, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's **start position**, the **destination** and the **maze**, find the shortest distance for the ball to stop at the destination. The distance is defined by the number of **empty spaces** traveled by the ball from the start position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

Example 1

```
Input 1: a maze represented by a 2D array

0 0 1 0 0
0 0 0 0 0
0 0 0 1 0
1 1 0 1 1
0 0 0 0 0

Input 2: start coordinate (rowStart, colStart) = (0, 4)
Input 3: destination coordinate (rowDest, colDest) = (4, 4)

Output: 12
Explanation: One shortest way is : left -> down -> left -> down -> right -> down ->
 right.
             The total distance is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.
```

Example 2

```
Input 1: a maze represented by a 2D array

0 0 1 0 0
0 0 0 0 0
0 0 0 1 0
1 1 0 1 1
0 0 0 0 0

Input 2: start coordinate (rowStart, colStart) = (0, 4)
Input 3: destination coordinate (rowDest, colDest) = (3, 2)

Output: -1
Explanation: There is no way for the ball to stop at the destination.
```

Note:

1. There is only one ball and one destination in the maze.
2. Both the ball and the destination exist on an empty space, and they will not be at the same position initially.
3. The given maze does not contain border (like the red rectangle in the example pictures), but you could assume the border of the maze are all walls.
4. The maze contains at least 2 empty spaces, and both the width and height of the maze won't exceed 100.

Solution 1

Solution of *The Maze*: https://discuss.leetcode.com/topic/77471/easy-understanding-java-bfs-solution
Solution of *The Maze III*: https://discuss.leetcode.com/topic/77474/similar-to-the-maze-ii-easy-understanding-java-bfs-solution

We need to use `PriorityQueue` instead of standard queue, and record the minimal length of each point.

```
public class Solution {
    class Point {
        int x,y,l;
        public Point(int _x, int _y, int _l) {x=_x;y=_y;l=_l;}
    }
    public int shortestDistance(int[][] maze, int[] start, int[] destination) {
        int m=maze.length, n=maze[0].length;
        int[][] length=new int[m][n]; // record length
        for (int i=0;i<m*n;i++) length[i/n][i%n]=Integer.MAX_VALUE;
        int[][] dir=new int[][] {{-1,0},{0,1},{1,0},{0,-1}};
        PriorityQueue<Point> list=new PriorityQueue<>((o1,o2)->o1.l-o2.l); // usi
ng priority queue
        list.offer(new Point(start[0], start[1], 0));
        while (!list.isEmpty()) {
            Point p=list.poll();
            if (length[p.x][p.y]<=p.l) continue; // if we have already found a ro
ute shorter
            length[p.x][p.y]=p.l;
            for (int i=0;i<4;i++) {
                int xx=p.x, yy=p.y, l=p.l;
                while (xx>=0 && xx<m && yy>=0 && yy<n && maze[xx][yy]==0) {
                    xx+=dir[i][0];
                    yy+=dir[i][1];
                    l++;
                }
                xx-=dir[i][0];
                yy-=dir[i][1];
                l--;
                list.offer(new Point(xx, yy, l));
            }
        }
        return length[destination[0]][destination[1]]==Integer.MAX_VALUE?-1:lengt
h[destination[0]][destination[1]];
    }
}
```

written by ckcz123 original link here

Solution 2

My BFS using queue. Use matrix to store the distance. If smaller distance is found, need to process the position[i,j] one more time.

```
class Solution {
public:
    int shortestDistance(vector<vector<int>>& maze, vector<int>& start, vector<in
t>& destination) {
 int m = maze.size();
 int n = maze[0].size();
 int minDist = INT_MAX;

 vector<vector<int>> dists(m, vector<int>(n, -1));
 queue<pair<int, int>> q;

 vector<pair<int, int>> incr = { { 1,0 },{ 0,-1 },{ 0,1 },{ -1,0 } };

 q.push({ start[0], start[1] });
 dists[start[0]][start[1]] = 0;

 while (!q.empty())
 {
  auto curr = q.front();
  q.pop();
  int x = curr.first;
  int y = curr.second;
  int dist = dists[x][y];

  for (int k = 0; k < 4; ++k)
  {
   int i = x;
   int j = y;
   int step = 0;
   int d_i = incr[k].first;
   int d_j = incr[k].second;
   int tempMin = INT_MAX;

   while (i + d_i < m && i + d_i >= 0 && j + d_j >= 0 && j + d_j <n && maze[i + d
_i][j + d_j] == 0)
   {
    ++step;
    i += incr[k].first;
    j += incr[k].second;

   }
   if (dists[i][j] == -1) // visited first time
   {
    dists[i][j] = dist + step;
    q.push({ i,j });
   }
   else
   {
    if (dists[i][j] > dist + step) // not the first time, but generate smaller di
st, process one more time
    {
```

```
      dists[i][j] = dist + step;
      q.push({ i,j });
     }
    }
   }
  }

  return dists[destination[0]][destination[1]];
     }
};
```

written by xiapeggy original link here

Solution 3

The idea is similar to the Dijkstra's shortest Path Algorithm.

```
public class Solution {
    private int[][] maze;
    private int[][] minSteps;//memorize the minimus steps to reach each position
    private int[][] dirs={{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    public int shortestDistance(int[][] maze, int[] start, int[] destination) {
        this.maze=maze;
        this.minSteps=new int[maze.length][maze[0].length];
        /*Initiallize minSteps matrix*/
        for(int i=0; i<maze.length; i++){
            for(int j=0; j<maze[0].length; j++){
                minSteps[i][j]=Integer.MAX_VALUE;
            }
        }
        /*Optimization: check if the destination is impossible to Reach*/
        boolean desL=canRoll(destination[0], destination[1], dirs[0]);
        boolean desR=canRoll(destination[0], destination[1], dirs[1]);
        boolean desD=canRoll(destination[0], destination[1], dirs[2]);
        boolean desU=canRoll(destination[0], destination[1], dirs[3]);
        if(desL && desR && desD && desU) return -1; //all neighbors are walls
        else if(!(desL||desR||desD||desU)) return -1; //all neighbors are empty s
paces
        else if(desL && desR && !desU && !desD) return -1; //two opposite neigbho
rs are walls, and the other two are empty spaces
        else if(!desL && !desR && desU && desD) return -1;//two opposite neigbhor
s are walls, and the other two are empty spaces

        minSteps[start[0]][start[1]]=0;
        /*BFS; Optimization: use PriorityQueue based on the steps instead of Queu
e*/
        PriorityQueue<Position> pq=new PriorityQueue<>();
        pq.offer(new Position(start[0], start[1], 0));
        while(!pq.isEmpty()){
            Position pos=pq.poll();
            /*optimization: if the destination is at the head of the queue, we are
done*/
            if(pos.r==destination[0] && pos.c==destination[1]) return pos.steps;
            for(int[] dir: dirs){
                int r=pos.r, c=pos.c, currSteps=0;
                while(canRoll(r, c, dir)){
                    r+=dir[0];
                    c+=dir[1];
                    currSteps++;
                }
                int totalSteps=pos.steps+currSteps;
                if(totalSteps<minSteps[r][c] && totalSteps<minSteps[destination[0
]][destination[1]]){
                    minSteps[r][c]=totalSteps;
                    pq.offer(new Position(r, c, totalSteps));
                }
            }
        }
        /*May not be able to reach the destination*/
```

```
        /*May not be able to reach the destination*/
        return minSteps[destination[0]][destination[1]]==Integer.MAX_VALUE? -1: m
inSteps[destination[0]][destination[1]];
    }

    private boolean canRoll(int r, int c, int[] dir){
        r+=dir[0];
        c+=dir[1];
        if(r<0 || c<0 || r>=maze.length || c>=maze[0].length || maze[r][c]==1) re
turn false;
        return true;
    }
}

class Position implements Comparable<Position>{
    public int r;
    public int c;
    public int steps;

    public Position(int r, int c, int s){
        this.r=r;
        this.c=c;
        this.steps=s;
    }
    @Override
    public int compareTo(Position other){
        return this.steps-other.steps;
    }
}
```

written by ydeng original link here

From Leetcoder.

Relative Ranks

Given scores of **N** athletes, find their relative ranks and the people with the top three highest scores, who will be awarded medals: "Gold Medal", "Silver Medal" and "Bronze Medal".

Example 1:

```
Input: [5, 4, 3, 2, 1]
Output: ["Gold Medal", "Silver Medal", "Bronze Medal", "4", "5"]
Explanation: The first three athletes got the top three highest scores, so they got
 "Gold Medal", "Silver Medal" and "Bronze Medal".
For the left two athletes, you just need to output their relative ranks according t
o their scores.
```

Note:

1. N is a positive integer and won't exceed 10,000.
2. All the scores of athletes are guaranteed to be unique.

Solution 1

Basically this question is to find out the `score` -> `ranking` mapping. The easiest way is to sort those `scores` in `nums`. But we will lose their original order. We can create (`score` , `original index`) pairs and sort them by `score` decreasingly. Then we will have `score` -> `ranking` (new index) mapping and we can use `original index` to create the result.

Time complexity: O(NlgN). Space complexity: O(N). N is the number of scores.

```
Example:

nums[i]    : [10, 3, 8, 9, 4]
pair[i][0] : [10, 3, 8, 9, 4]
pair[i][1] : [ 0, 1, 2, 3, 4]

After sort:
pair[i][0] : [10, 9, 8, 4, 3]
pair[i][1] : [ 0, 3, 2, 4, 1]
```

```java
public class Solution {
    public String[] findRelativeRanks(int[] nums) {
        int[][] pair = new int[nums.length][2];

        for (int i = 0; i < nums.length; i++) {
            pair[i][0] = nums[i];
            pair[i][1] = i;
        }

        Arrays.sort(pair, (a, b) -> (b[0] - a[0]));

        String[] result = new String[nums.length];

        for (int i = 0; i < nums.length; i++) {
            if (i == 0) {
                result[pair[i][1]] = "Gold Medal";
            }
            else if (i == 1) {
                result[pair[i][1]] = "Silver Medal";
            }
            else if (i == 2) {
                result[pair[i][1]] = "Bronze Medal";
            }
            else {
                result[pair[i][1]] = (i + 1) + "";
            }
        }

        return result;
    }
}
```

Also we can use an one dimension array. This will save a little bit space but space

complexity is still O(n).

```
public class Solution {
    public String[] findRelativeRanks(int[] nums) {
        Integer[] index = new Integer[nums.length];

        for (int i = 0; i < nums.length; i++) {
            index[i] = i;
        }

        Arrays.sort(index, (a, b) -> (nums[b] - nums[a]));

        String[] result = new String[nums.length];

        for (int i = 0; i < nums.length; i++) {
            if (i == 0) {
                result[index[i]] = "Gold Medal";
            }
            else if (i == 1) {
                result[index[i]] = "Silver Medal";
            }
            else if (i == 2) {
                result[index[i]] = "Bronze Medal";
            }
            else {
                result[index[i]] = (i + 1) + "";
            }
        }

        return result;
    }
}
```

written by shawngao original link here

Solution 2

Use a dictionary mapping scores to ranks.

```
def findRelativeRanks(self, nums):
    sort = sorted(nums)[::-1]
    rank = ["Gold Medal", "Silver Medal", "Bronze Medal"] + map(str, range(4, len
(nums) + 1))
    return map(dict(zip(sort, rank)).get, nums)
```

written by StefanPochmann original link here

Solution 3

```
class Solution {
public:
    vector<string> findRelativeRanks(vector<int>& nums) {
        vector<int> rank;
        for(int i=0; i<nums.size(); ++i) rank.push_back(i);

        sort(rank.begin(), rank.end(), [&](int a, int b){return nums[a] > nums[b]
;});
        vector<string> ranks(nums.size());

        for(int i=3; i<nums.size(); ++i){
            ranks[rank[i]] = std::to_string(i+1);
        }

        if(nums.size() > 0) ranks[rank[0]] = "Gold Medal";
        if(nums.size() > 1) ranks[rank[1]] = "Silver Medal";
        if(nums.size() > 2) ranks[rank[2]] = "Bronze Medal";

        return ranks;
    }
};
```

written by kevin36 original link here

From Leetcoder.

Most Frequent Subtree Sum

Given the root of a tree, you are asked to find the most frequent subtree sum. The subtree sum of a node is defined as the sum of all the node values formed by the subtree rooted at that node (including the node itself). So what is the most frequent subtree sum value? If there is a tie, return all the values with the highest frequency in any order.

Examples 1
Input:

```
  5
 /  \
2   −3
```

return [2, -3, 4], since all the values happen only once, return all of them in any order.
Examples 2
Input:

```
  5
 /  \
2   −5
```

return [2], since 2 happens twice, however -5 only occur once.
Note: You may assume the sum of values in any subtree is in the range of 32-bit signed integer.

Solution 1

For sake of saving time during contest, can't write so concise solution :)
Idea is `post-order` traverse the tree and get sum of every sub-tree, put `sum` to `count` mapping to a HashMap. Then generate result based on the HashMap.

```
public class Solution {
    Map<Integer, Integer> sumToCount;
    int maxCount;

    public int[] findFrequentTreeSum(TreeNode root) {
        maxCount = 0;
        sumToCount = new HashMap<Integer, Integer>();

        postOrder(root);

        List<Integer> res = new ArrayList<>();
        for (int key : sumToCount.keySet()) {
            if (sumToCount.get(key) == maxCount) {
                res.add(key);
            }
        }

        int[] result = new int[res.size()];
        for (int i = 0; i < res.size(); i++) {
            result[i] = res.get(i);
        }
        return result;
    }

    private int postOrder(TreeNode root) {
        if (root == null) return 0;

        int left = postOrder(root.left);
        int right = postOrder(root.right);

        int sum = left + right + root.val;
        int count = sumToCount.getOrDefault(sum, 0) + 1;
        sumToCount.put(sum, count);

        maxCount = Math.max(maxCount, count);

        return sum;
    }
}
```

written by shawngao original link here

Solution 2

I have used a hash-map `ctr` to count the sum occurrence.

I have wrote a sub function `countSubtreeSum` to travel through a tree and return the sum of the tree.

In `countSubtreeSum`, I will travel through the left node and the right node, calculate the sum of the tree, increment the counter `ctr`, and return the sum.

```
def findFrequentTreeSum(self, root):
      if root == None: return []

      def getSum(node):
          if node == None: return 0
          s = node.val + getSum(node.left) + getSum(node.right)
          c[s] += 1
          return s

      from collections import Counter
      c = Counter()
      getSum(root)
      frequent = max(c.values())
      return [s for s in c.keys() if c[s] == frequent]
```

Please upvote if it makes sense.

Update: I have changed a little to make the variable name shorter.

written by lee215 original link here

Solution 3

```
class Solution {
public:
    vector<int> findFrequentTreeSum(TreeNode* root) {
        unordered_map<int,int> counts;
        int maxCount = 0;
        countSubtreeSums(root, counts, maxCount);


        vector<int> maxSums;
        for(const auto& x :  counts){
            if(x.second == maxCount) maxSums.push_back(x.first);
        }
        return maxSums;
    }

    int countSubtreeSums(TreeNode *r, unordered_map<int,int> &counts, int& maxCou
nt){
        if(r == nullptr) return 0;

        int sum = r->val;
        sum += countSubtreeSums(r->left, counts, maxCount);
        sum += countSubtreeSums(r->right, counts, maxCount);
        ++counts[sum];
        maxCount = max(maxCount, counts[sum]);
        return sum;
    }
};
```

written by kevin36 original link here

From Leetcoder.

Find Bottom Left Tree Value

Given a binary tree, find the leftmost value in the last row of the tree.

Example 1:

```
Input:

    2
   / \
  1   3

Output:
1
```

Example 2:

```
Input:

        1
       / \
      2   3
     /   / \
    4   5   6
       /
      7

Output:
7
```

Note: You may assume the tree (i.e., the given root node) is not**NULL**.

Solution 1

Typical way to do binary tree level order traversal. Only additional step is to remember the `first` element of each level.

```
public class Solution {
    public int findLeftMostNode(TreeNode root) {
        if (root == null) return 0;

        int result = 0;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                if (i == 0) result = node.val;
                if (node.left != null) queue.add(node.left);
                if (node.right != null) queue.add(node.right);
            }
        }

        return result;
    }
}
```

written by shawngao original link here

Solution 2

```
public class Solution {
    int ans=0, h=0;
    public int findBottomLeftValue(TreeNode root) {
        findBottomLeftValue(root, 1);
        return ans;
    }
    public void findBottomLeftValue(TreeNode root, int depth) {
        if (h<depth) {ans=root.val;h=depth;}
        if (root.left!=null) findBottomLeftValue(root.left, depth+1);
        if (root.right!=null) findBottomLeftValue(root.right, depth+1);
    }
}
```

No global variables, 6ms (faster):

```
public class Solution {
    public int findBottomLeftValue(TreeNode root) {
        return findBottomLeftValue(root, 1, new int[]{0,0});
    }
    public int findBottomLeftValue(TreeNode root, int depth, int[] res) {
        if (res[1]<depth) {res[0]=root.val;res[1]=depth;}
        if (root.left!=null) findBottomLeftValue(root.left, depth+1, res);
        if (root.right!=null) findBottomLeftValue(root.right, depth+1, res);
        return res[0];
    }
}
```

written by ckcz123 original link here

Solution 3

Got the idea from @fallcreek. Doing BFS right to left means we can simply return the **last** node's value and don't have to keep track of the first node in the current tree row.

Python:

```
def findLeftMostNode(self, root):
    queue = [root]
    for node in queue:
        queue += filter(None, (node.right, node.left))
    return node.val
```

Java:

```
public int findLeftMostNode(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        root = queue.poll();
        if (root.right != null)
            queue.add(root.right);
        if (root.left != null)
            queue.add(root.left);
    }
    return root.val;
}
```

written by StefanPochmann original link here

From Leetcoder.

Find Largest Value in Each Tree Row

You need to find the largest value in each row of a binary tree.

Example:

```
Input:

          1
         / \
        3   2
       / \   \
      5   3   9

Output: [1, 3, 9]
```

Solution 1

```
def findValueMostElement(self, root):
    maxes = []
    row = [root]
    while any(row):
        maxes.append(max(node.val for node in row))
        row = [kid for node in row for kid in (node.left, node.right) if kid]
    return maxes
```

written by StefanPochmann original link here

Solution 2

Just a simple pre-order traverse idea. Use depth to expand result list size and put the max value in the appropriate position.

```
public class Solution {
    public List<Integer> largestValues(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        helper(root, res, 0);
        return res;
    }
    private void helper(TreeNode root, List<Integer> res, int d){
        if(root == null){
            return;
        }
       //expand list size
        if(d == res.size()){
            res.add(root.val);
        }
        else{
        //or set value
            res.set(d, Math.max(res.get(d), root.val));
        }
        helper(root.left, res, d+1);
        helper(root.right, res, d+1);
    }
}
```

written by Ryan777 original link here

Solution 3

Alright, two binary tree level order traversal problems in one contest. This time, mission is to find the `max` of each level...

```
public class Solution {
    public int[] findValueMostElement(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) return new int[0];

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            int size = queue.size();
            int max = Integer.MIN_VALUE;
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                max = Math.max(max, node.val);
                if (node.left != null) queue.add(node.left);
                if (node.right != null) queue.add(node.right);
            }
            res.add(max);
        }

        int[] result = new int[res.size()];
        for (int i = 0; i < res.size(); i++) {
            result[i] = res.get(i);
        }

        return result;
    }
}
```

written by shawngao original link here

From Leetcoder.

Longest Palindromic Subsequence

Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.

Example 1:
Input:

```
"bbbab"
```

Output:

```
4
```

One possible longest palindromic subsequence is "bbbb".
Example 2:
Input:

```
"cbbd"
```

Output:

```
2
```

One possible longest palindromic subsequence is "bb".

Solution 1

`dp[i][j]` : the longest palindromic subsequence's length of substring(i, j)
`State transition`:
`dp[i][j] = dp[i+1][j-1] + 2` if s.charAt(i) == s.charAt(j)
otherwise, `dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1])`
`Initialization`: `dp[i][i] = 1`

```
public class Solution {
    public int longestPalindromeSubseq(String s) {
        int[][] dp = new int[s.length()][s.length()];

        for (int i = s.length() - 1; i >= 0; i--) {
            dp[i][i] = 1;
            for (int j = i+1; j < s.length(); j++) {
                if (s.charAt(i) == s.charAt(j)) {
                    dp[i][j] = dp[i+1][j-1] + 2;
                } else {
                    dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
                }
            }
        }
        return dp[0][s.length()-1];
    }
}
```

Top bottom recursive method with memoization

```
public class Solution {
    public int longestPalindromeSubseq(String s) {
        return helper(s, 0, s.length() - 1, new Integer[s.length()][s.length()]);
    }

    private int helper(String s, int i, int j, Integer[][] memo) {
        if (memo[i][j] != null) {
            return memo[i][j];
        }
        if (i > j)      return 0;
        if (i == j)     return 1;

        if (s.charAt(i) == s.charAt(j)) {
            memo[i][j] = helper(s, i + 1, j - 1, memo) + 2;
        } else {
            memo[i][j] = Math.max(helper(s, i + 1, j, memo), helper(s, i, j - 1,
memo));
        }
        return memo[i][j];
    }
}
```

written by tankztc original link here

Solution 2

Idea:
dp[i][j] = longest palindrome subsequence of s[i to j].
If s[i] == s[j], dp[i][j] = 2 + dp[i+1][j - 1]
Else, dp[i][j] = max(dp[i+1][j], dp[i][j-1])

Rolling array O(2n) space

```
class Solution(object):
    def longestPalindromeSubseq(self, s):
        """
        :type s: str
        :rtype: int
        """
        n = len(s)
        dp = [[1] * 2 for _ in range(n)]
        for j in xrange(1, len(s)):
            for i in reversed(xrange(0, j)):
                if s[i] == s[j]:
                    dp[i][j%2] = 2 + dp[i + 1][(j - 1)%2] if i + 1 <= j - 1 else
2
                else:
                    dp[i][j%2] = max(dp[i + 1][j%2], dp[i][(j - 1)%2])
        return dp[0][(n-1)%2]
```

Further improve space to O(n)

```
class Solution(object):
    def longestPalindromeSubseq(self, s):
        """
        :type s: str
        :rtype: int
        """
        n = len(s)
        dp = [1] * n
        for j in xrange(1, len(s)):
            pre = dp[j]
            for i in reversed(xrange(0, j)):
                tmp = dp[i]
                if s[i] == s[j]:
                    dp[i] = 2 + pre if i + 1 <= j - 1 else 2
                else:
                    dp[i] = max(dp[i + 1], dp[i])
                pre = tmp
        return dp[0]
```

written by jedihy original link here

Solution 3

1. O(2^n) Brute force. If the two ends of a string are the same, then they must be included in the longest palindrome subsequence. Otherwise, both ends cannot be included in the longest palindrome subsequence.

```
    int longestPalindromeSubseq(string s) {
        return longestPalindromeSubseq(0,s.size()-1,s);
    }
    int longestPalindromeSubseq(int l, int r, string &s) {
        if(l==r) return 1;
        if(l>r) return 0;  //happens after "aa"
        return s[l]==s[r] ? 2 + longestPalindromeSubseq(l+1,r-1, s) :
            max(longestPalindromeSubseq(l+1,r, s),longestPalindromeSubseq(l,r-1,
s));
    }
```

2. O(n^2) Memoization

```
    int longestPalindromeSubseq(string s) {
        int n = s.size();
        vector<vector<int>> mem(n,vector<int>(n));
        return longestPalindromeSubseq(0,n-1, s,mem);
    }
    int longestPalindromeSubseq(int l, int r, string &s, vector<vector<int>>& mem)
{
        if(l==r) return 1;
        if(l>r) return 0;
        if(mem[l][r]) return mem[l][r];
        return mem[l][r] = s[l]==s[r] ? 2 + longestPalindromeSubseq(l+1,r-1, s,me
m) :
            max(longestPalindromeSubseq(l+1,r, s,mem),longestPalindromeSubseq(l,r
-1, s,mem));

    }
```

3. O(n^2) dp

```
    int longestPalindromeSubseq(string s) {
        int n = s.size();
        vector<vector<int>> dp(n+1,vector<int>(n));
        for(int i=0;i<n;i++) dp[1][i]=1;
        for(int i=2;i<=n;i++) //length
            for(int j=0;j<n-i+1;j++) {//start index
                dp[i][j] = s[j]==s[i+j-1]?2+dp[i-2][j+1]:max(dp[i-1][j],dp[i-1][j
+1]);
        return dp[n][0];
    }
```

4. O(n^2) time, O(n) space dp. In #3, the current row is computed from the previous 2 rows only. So we don't need to keep all the rows.

```
    int longestPalindromeSubseq(string s) {
        int n = s.size();
        vector<int> v0(n), v1(n,1), v(n), *i_2=&v0, *i_1=&v1, *i_=&v;
        for(int i=2;i<=n;i++) {//length
            for(int j=0;j<n-i+1;j++)//start index
                i_->at(j) = s[j]==s[i+j-1]?2+i_2->at(j+1):max(i_1->at(j),i_1->at(
j+1));
            swap(i_1,i_2);
            swap(i_1,i_); //rotate i_2, i_1, i_
        }
        return i_1->at(0);
    }
```

written by yu6 original link here

From Leetcoder.

Super Washing Machines

You have **n** super washing machines on a line. Initially, each washing machine has some dresses or is empty.

For each **move**, you could choose **any m** (1 ≤ m ≤ n) washing machines, and pass **one dress** of each washing machine to one of its adjacent washing machines**at the same time** .

Given an integer array representing the number of dresses in each washing machine from left to right on the line, you should find the **minimum number of moves** to make all the washing machines have the same number of dresses. If it is not possible to do it, return -1.

Example1

```
Input: [1,0,5]

Output: 3

Explanation:
1st move:    1     0     1     1     4
2nd move:    1     2     1     3
3rd move:    2     1     2     2     2
```

Example2

```
Input: [0,3,0]

Output: 2

Explanation:
1st move:    0     1     2     0
2nd move:    1     2 --> 0    =>    1     1     1
```

Example3

```
Input: [0,2,0]

Output: -1

Explanation:
It's impossible to make all the three washing machines have the same number of dres
ses.
```

Note:

1. The range of n is [1, 10000].
2. The range of dresses number in a super washing machine is [0, 1e5].

Solution 1

First we check the sum of dresses in all machines. if that number cannot be divided by count of machines, there is no solution.

Otherwise, we can always transfer a dress from one machine to another, one at a time until every machines reach the same number, so there must be a solution. In this way, the total actions is sum of operations on every machine.

Since we can operate several machines at the same time, the minium number of moves is the maximum number of necessary operations on every machine.

For a single machine, necessary operations is to transfer dresses from one side to another until sum of both sides and itself reaches the average number. We can calculate (required dresses) - (contained dresses) of each side as L and R:

L > 0 && R > 0: both sides lacks dresses, and we can only export one dress from current machines at a time, so result is abs(L) + abs(R)
L < 0 && R < 0: both sides contains too many dresses, and we can import dresses from both sides at the same time, so result is max(abs(L), abs(R))
L < 0 && R > 0 or L >0 && R < 0: the side with a larger absolute value will import/export its extra dresses from/to current machine or other side, so result is max(abs(L), abs(R))

For example, [1, 0, 5], average is 2
for 1, L = 0 * 2 - 0 = 0, R = 2 * 2 - 5= -1, result = 1
for 0, L = 1 * 2 - 1= 1, R = 1 * 2 - 5 = -3, result = 3
for 5, L = 2 * 2 - 1= 3, R = 0 * 2 - 0= 0, result = 3
so minium moves is 3

```
class Solution {
public:
    int findMinMoves(vector<int>& machines) {
        int len = machines.size();
        vector<int> sum(len + 1, 0);
        for (int i = 0; i < len; ++i)
            sum[i + 1] = sum[i] + machines[i];

        if (sum[len] % len) return -1;

        int avg = sum[len] / len;
        int res = 0;
        for (int i = 0; i < len; ++i)
        {
            int l = i * avg - sum[i];
            int r = (len - i - 1) * avg - (sum[len] - sum[i] - machines[i]);

            if (l > 0 && r > 0)
                res = std::max(res, std::abs(l) + std::abs(r));
            else
                res = std::max(res, std::max(std::abs(l), std::abs(r)));
        }
        return res;
    }
};
```

written by Mrsuyi original link here

Solution 2

```
public class Solution {
    public int findMinMoves(int[] machines) {
        int total = 0;
        for(int i: machines) total+=i;
        if(total%machines.length!=0) return -1;
        int avg = total/machines.length, cnt = 0, max = 0;
        for(int load: machines){
            cnt += load-avg; //load-avg is "gain/lose"
            max = Math.max(Math.max(max, Math.abs(cnt)), load-avg);
        }
        return max;
    }
}
```

Let me use an example to briefly explain this. For example, your machines[] is [0,0,11,5]. So your total is 16 and the target value for each machine is 4. Convert the machines array to a kind of gain/lose array, we get: [-4,-4,7,1]. Now what we want to do is go from the first one and try to make all of them 0.
To make the 1st machines 0, you need to give all its "load" to the 2nd machines.
So we get: [0,-8,7,1]
then: [0,0,-1,1]
lastly: [0,0,0,0], done.
You don't have to worry about the details about how these machines give load to each other. In this process, the least steps we need to eventually finish this process is determined by the peak of abs(cnt) and the max of "gain/lose" array. In this case, the peak of abs(cnt) is 8 and the max of gain/lose array is 7. So the result is 8.

Some other example:
machines: [0,3,0]; gain/lose array: [-1,2,-1]; max = 2, cnt = 0, -1, 1, 0, its abs peak is 1. So result is 2.
machines: [1,0,5]; gain/lose array: [-1,-2,3]; max = 3, cnt = 0, -1, -3, 0, its abs peak is 3. So result is 3.

written by Chidong original link here

Solution 3

```java
public class Solution {
    public int findMinMoves(int[] machines) {
        int total = 0, target = 0, result = 0, n = machines.length;
        for (int d : machines) total += d;
        if (total == 0) return 0;
        if (total % n != 0) return -1;
        target = total / n;

        int[] move = new int[n];
        for (int i = 0; i < n - 1; i++) {
            if (machines[i] > target) {
                move[i] += machines[i] - target;
                machines[i + 1] += machines[i] - target;
                machines[i] = target;
                result = Math.max(result, move[i]);
            }
            else {
                move[i + 1] = target - machines[i];
                machines[i + 1] -= target - machines[i];
                machines[i] = target;
                result = Math.max(result, move[i + 1]);
            }
        }

        return result;
    }
}
```

written by shawngao original link here

From Leetcoder.

Detect Capital

Given a word, you need to judge whether the usage of capitals in it is right or not.

We define the usage of capitals in a word to be right when one of the following cases holds:

1. All letters in this word are capitals, like "USA".
2. All letters in this word are not capitals, like "leetcode".
3. Only the first letter in this word is capital if it has more than one letter, like "Google".

Otherwise, we define that this word doesn't use capitals in a right way.

Example 1:

```
Input: "USA"
Output: True
```

Example 2:

```
Input: "FlaG"
Output: False
```

Note: The input will be a non-empty word consisting of uppercase and lowercase latin letters.

Solution 1

Regex again.

```
public boolean detectCapitalUse(String word) {
    return word.matches("[A-Z]+|[a-z]+|[A-Z][a-z]+");
}
```

written by lixx2100 original link here

Solution 2

```
def detectCapitalUse(self, word):
    return word.isupper() or word.islower() or word.istitle()
```

written by StefanPochmann original link here

Solution 3

```
public class Solution {
    public boolean detectCapitalUse(String word) {
        int cnt = 0;
        for(char c: word.toCharArray()) if('Z' - c >= 0) cnt++;
        return ((cnt==0 || cnt==word.length()) || (cnt==1 && 'Z' - word.charAt(0)
>=0));
    }
}
```

written by Chidong original link here

From Leetcoder.

Continuous Subarray Sum

Given a list of **non-negative** numbers and a target **integer** k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to the multiple of **k**, that is, sums up to n*k where n is also an **integer**.

Example 1:

```
Input: [23, 2, 4, 6, 7],  k=6
Output: True
Explanation: Because [2, 4] is a continuous subarray of size 2 and sums up to 6.
```

Example 2:

```
Input: [23, 2, 6, 4, 7],  k=6
Output: True
Explanation: Because [23, 2, 6, 4, 7] is an continuous subarray of size 5 and sums
up to 42.
```

Note:

1. The length of the array won't exceed 10,000.
2. You may assume the sum of all the numbers is in the range of a signed 32-bit integer.

Solution 1

We iterate through the input array exactly once, keeping track of the running sum mod k of the elements in the process. If we find that a running sum value at index j has been previously seen before in some earlier index i in the array, then we know that the sub-array (i,j] contains a desired sum.

```
public boolean checkSubarraySum(int[] nums, int k) {
    Map<Integer, Integer> map = new HashMap<Integer, Integer>(){{put(0,-1);}};;
    int runningSum = 0;
    for (int i=0;i<nums.length;i++) {
        runningSum += nums[i];
        if (k != 0) runningSum %= k;
        Integer prev = map.get(runningSum);
        if (prev != null) {
            if (i - prev > 1) return true;
        }
        else map.put(runningSum, i);
    }
    return false;
}
```

written by compton_scatter original link here

Solution 2

- `if k == 0`
  If there are two continuous zeros in `nums`, return `True`

  Time O(n).

- `if n >= 2k and k > 0`
  There will be at least three numbers in `sum` with the same remainder divided by `k`. So I can return True without any extra calculation.

  Time O(1).

- `if n < 2k and k > 0`
  If I can find two numbers in `sum` with the same remainder divided by `k` and the distance of them is greater than or equal to 2， return `True`.

  Time O(n) <= O(k).

- `k < 0`
  same as `k > 0`.

```
class Solution(object):
    def checkSubarraySum(self, nums, k):


        if k == 0:
            # if two continuous zeros in nums, return True
            # time O(n)
            for i in range(0, len(nums) - 1):
                if nums[i] == 0 and nums[i+1] == 0:
                    return True
            return False

        k = abs(k)
        if len(nums) >= k * 2:
            return True

        #if n >= 2k: return True
        #if n < 2k:  time O(n) is O(k)

        sum = [0]
        for x in nums:
            sum.append((sum[-1] + x) % k)

        Dict = {}
        for i in range(0, len(sum)):
            if Dict.has_key(sum[i]):
                if i - Dict[sum[i]] > 1:
                    return True
            else:
                Dict[sum[i]] = i

        return False
```

written by zym_ustc original link here

Solution 3

This problem contributed a lot of bugs to my contest score... Let's read the description again, pay attention to `red` sections:

Given a list of `non-negative` numbers and a target integer k, write a function to check if the array has a `continuous subarray` of size `at least 2` that sums up to the `multiple` of k, that is, sums up to n*k where n is also an `integer`.

Some `damn it!` test cases:

1. [0], 0 -> false;
2. [5, 2, 4], 5 -> false;
3. [0, 0], 100 -> true;
4. [1,5], -6 -> true;
   etc...

```
public class Solution {
    public boolean checkSubarraySum(int[] nums, int k) {
        // Since the size of subarray is at least 2.
        if (nums.length <= 1) return false;
        // Two continuous "0" will form a subarray which has sum = 0. 0 * k == 0
will always be true.
        for (int i = 0; i < nums.length - 1; i++) {
            if (nums[i] == 0 && nums[i + 1] == 0) return true;
        }

        // At this point, k can't be "0" any longer.
        if (k == 0) return false;
        // Let's only check positive k. Because if there is a n makes n * k = sum
, it is always true -n * -k = sum.
        if (k < 0) k = -k;

        Set<Integer> sums = new HashSet<>();
        int sum = 0;
        sums.add(0);

        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];

            if (i > 0) {
                // Validate from the biggest possible n * k to k
                for (int j = (sum / k) * k; j >= k; j -= k) {
                    if (sums.contains(sum - j)) return true;
                }
            }

            sums.add(sum);
        }

        return false;
    }
}
```

written by shawngao original link here

From Leetcoder.

Longest Word in Dictionary through Deleting

Given a string and a string dictionary, find the longest string in the dictionary that can be formed by deleting some characters of the given string. If there are more than one possible results, return the longest word with the smallest lexicographical order. If there is no possible result, return the empty string.

Example 1:

```
Input:
s = "abpcplea", d = ["ale","apple","monkey","plea"]

Output:
"apple"
```

Example 2:

```
Input:
s = "abpcplea", d = ["a","b","c"]

Output:
"a"
```

Note:

1. All the strings in the input will only contain lower-case letters.
2. The size of the dictionary won't exceed 1,000.
3. The length of all the strings in the input won't exceed 1,000.

Solution 1

We sort the input dictionary by longest length and lexicography. Then, we iterate through the dictionary exactly once. In the process, the first dictionary word in the sorted dictionary which appears as a subsequence in the input string s must be the desired solution.

```
public String findLongestWord(String s, List<String> d) {
    Collections.sort(d, (a,b) -> a.length() != b.length() ? -Integer.compare(a.le
ngth(), b.length()) :  a.compareTo(b));
    for (String dictWord : d) {
        int i = 0;
        for (char c : s.toCharArray())
            if (i < dictWord.length() && c == dictWord.charAt(i)) i++;
        if (i == dictWord.length()) return dictWord;
    }
    return "";
}
```

An alternate, more efficient solution which avoids sorting the dictionary:

```
public String findLongestWord(String s, List<String> d) {
    String longest = "";
    for (String dictWord : d) {
        int i = 0;
        for (char c : s.toCharArray())
            if (i < dictWord.length() && c == dictWord.charAt(i)) i++;

        if (i == dictWord.length() && dictWord.length() >= longest.length())
            if (dictWord.length() > longest.length() || dictWord.compareTo(longes
t) < 0)
                longest = dictWord;
    }
    return longest;
}
```

Time Complexity: O(nk), where n is the length of string s and k is the number of words in the dictionary.

written by compton_scatter original link here

Solution 2

```
def findLongestWord(self, s, d):
    def isSubsequence(x):
        it = iter(s)
        return all(c in it for c in x)
    return max(sorted(filter(isSubsequence, d)) + [''], key=len)
```

More efficient version (no sorting):

```
def findLongestWord(self, s, d):
    def isSubsequence(x):
        it = iter(s)
        return all(c in it for c in x)
    return min(filter(isSubsequence, d) + [''], key=lambda x: (-len(x), x))
```

Different style:

```
def findLongestWord(self, s, d):
    best = ''
    for x in d:
        if (-len(x), x) < (-len(best), best):
            it = iter(s)
            if all(c in it for c in x):
                best = x
    return best
```

Optimized as suggested by @easton042, testing from longest to shortest and returning the first valid one without testing the rest:

```
def findLongestWord(self, s, d):
    def isSubsequence(x):
        it = iter(s)
        return all(c in it for c in x)
    d.sort(key=lambda x: (-len(x), x))
    return next(itertools.ifilter(isSubsequence, d), '')
```

Or:

```
def findLongestWord(self, s, d):
    for x in sorted(d, key=lambda x: (-len(x), x)):
        it = iter(s)
        if all(c in it for c in x):
            return x
    return ''
```

And taking that even further by not sorting unnecessarily much:

```python
def findLongestWord(self, s, d):
    heap = [(-len(word), word) for word in d]
    heapq.heapify(heap)
    while heap:
        word = heapq.heappop(heap)[1]
        it = iter(s)
        if all(c in it for c in word):
            return word
    return ''
```

written by StefanPochmann original link here

Solution 3

I think there is no need to sort the dic, just iterate the dic and test whether the word is satisfied and whether we need update our answer.

```
    string findLongestWord(string s, vector<string>& d) {
        string ans;
        for (int i = 0; i < d.size(); i++) {
            int pi = 0, pj = 0;
            for (; pi < s.size() && pj < d[i].size(); pi++) {
                pj += s[pi] == d[i][pj];
            }
            if (pj == d[i].size() && (ans.size() < d[i].size() || (ans.size() ==
d[i].size() && ans > d[i])))
                ans = d[i];
        }
        return ans;
    }
```

written by saxion original link here

From Leetcoder.

Contiguous Array

Given a binary array, find the maximum length of a contiguous subarray with equal number of 0 and 1.

Example 1:

```
Input: [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with equal number of 0 and 1
.
```

Example 2:

```
Input: [0,1,0]
Output: 2
Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number
of 0 and 1.
```

Note: The length of the given binary array will not exceed 50,000.

Solution 1

The idea is to change `0` in the original array to `-1`. Thus, if we find `SUM[i, j] == 0` then we know there are even number of `-1` and `1` between index `i` and `j`. Also put the `sum` to `index` mapping to a HashMap to make search faster.

```
public class Solution {
    public int findMaxLength(int[] nums) {
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == 0) nums[i] = -1;
        }

        Map<Integer, Integer> sumToIndex = new HashMap<>();
        sumToIndex.put(0, -1);
        int sum = 0, max = 0;

        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];
            if (sumToIndex.containsKey(sum)) {
                max = Math.max(max, i - sumToIndex.get(sum));
            }
            else {
                sumToIndex.put(sum, i);
            }
        }

        return max;
    }
}
```

written by shawngao original link here

Solution 2

My idea is very similar to others, but let me try to explain it more visually. My thought was inspired by 121. Best Time to Buy and Sell Stock.

Let's have a variable `count` initially equals 0 and traverse through `nums`. Every time we meet a 0, we decrease `count` by 1, and increase `count` by 1 when we meet 1. It's pretty easy to conclude that we have a contiguous subarray with equal number of 0 and 1 when `count` equals 0.

What if we have a sequence `[0, 0, 0, 0, 1, 1]`? the maximum length is 4, the `count` starting from 0, will equal -1, -2, -3, -4, -3, -2, and won't go back to 0 again. But wait, the longest subarray with equal number of 0 and 1 started and ended when `count` equals -2. We can plot the changes of `count` on a graph, as shown below. Point (0,0) indicates the initial value of `count` is 0, so we count the sequence starting from index 1. The longest subarray is from index 2 to 6.

From above illustration, we can easily understand that two points with the same y-axis value indicates the sequence between these two points has equal number of 0 and 1.

Another example, sequence `[0, 0, 1, 0, 0, 0, 1, 1]`, as shown below,

There are 3 points have the same y-axis value -2. So subarray from index 2 to 4 has equal number of 0 and 1, and subarray from index 4 to 8 has equal number of 0 and 1. We can add them up to form the longest subarray from index 2 to 8, so the maximum length of the subarray is 8 - 2 = 6.

Yet another example, sequence [0, 1, 1, 0, 1, 1, 1, 0], as shown below. The longest subarray has the y-axis value of 0.

To find the maximum length, we need a dict to store the value of `count` (as the key) and its associated index (as the value). We only need to save a `count` value and its index at the first time, when the same `count` values appear again, we use the new index subtracting the old index to calculate the length of a subarray. A variable `max_length` is used to to keep track of the current maximum length.

```python
class Solution(object):
    def findMaxLength(self, nums):
        count = 0
        max_length=0
        table = {0: 0}
        for index, num in enumerate(nums, 1):
            if num == 0:
                count -= 1
            else:
                count += 1

            if count in table:
                max_length = max(max_length, index - table[count])
            else:
                table[count] = index

        return max_length
```

written by NoAnyLove original link here

Solution 3

Keeping track of the balance (number of ones minus number of zeros) and storing the first index where each balance occurred.

Python

Keeping the balance in units of 0.5 which makes the update expression short (not that `num * 2 - 1` or `1 if num else -1` would be terribly long):

```
def findMaxLength(self, nums):
    index = {0: -1}
    balance = maxlen = 0
    for i, num in enumerate(nums):
        balance += num - 0.5
        maxlen = max(maxlen, i - index.setdefault(balance, i))
    return maxlen
```

Just for fun as an ugly one-liner:

```
def findMaxLength(self, nums):
    return reduce(lambda(f,b,m),(i,x):(f,b+x-.5,max(m,i-f.setdefault(b+x-.5,i))),
enumerate(nums),({0:-1},0,0))[2]
```

Java

Using `putIfAbsent` so I only need one map function call per number.

```
public int findMaxLength(int[] nums) {
    Map<Integer, Integer> index = new HashMap<>();
    index.put(0, -1);
    int balance = 0, maxlen = 0;
    for (int i = 0; i < nums.length; i++) {
        balance += nums[i] * 2 - 1;
        Integer first = index.putIfAbsent(balance, i);
        if (first != null)
            maxlen = Math.max(maxlen, i - first);
    }
    return maxlen;
}
```

Could avoid using `Math.max` like this:

```
        if (first != null && i - first > maxlen)
            maxlen = i - first;
```

written by StefanPochmann original link here

Beautiful Arrangement

Suppose you have **N** integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these **N** numbers successfully if one of the following is true for the i_{th} position ($1 \le i \le N$) in this array:

1. The number at the i_{th} position is divisible by **i**.
2. **i** is divisible by the number at the i_{th} position.

Now given N, how many beautiful arrangements can you construct?

Example 1:

```
Input: 2
Output: 2
Explanation:

The first beautiful arrangement is [1, 2]:

Number at the 1st position (i=1) is 1, and 1 is divisible by i (i=1).

Number at the 2nd position (i=2) is 2, and 2 is divisible by i (i=2).

The second beautiful arrangement is [2, 1]:

Number at the 1st position (i=1) is 2, and 2 is divisible by i (i=1).

Number at the 2nd position (i=2) is 1, and i (i=2) is divisible by 1.
```

Note:

1. **N** is a positive integer and will not exceed 15.

Solution 1

Just try every possible number at each position...

```
public class Solution {
    int count = 0;

    public int countArrangement(int N) {
        if (N == 0) return 0;
        helper(N, 1, new int[N + 1]);
        return count;
    }

    private void helper(int N, int pos, int[] used) {
        if (pos > N) {
            count++;
            return;
        }

        for (int i = 1; i <= N; i++) {
            if (used[i] == 0 && (i % pos == 0 || pos % i == 0)) {
                used[i] = 1;
                helper(N, pos + 1, used);
                used[i] = 0;
            }
        }
    }
}
```

written by shawngao original link here

Solution 2

```
// By lovellp
// Time: 6ms
class Solution {
public:
    int countArrangement(int N) {
        vector<int> vs;
        for (int i=0; i<N; ++i) vs.push_back(i+1);
        return counts(N, vs);
    }
    int counts(int n, vector<int>& vs) {
        if (n <= 0) return 1;
        int ans = 0;
        for (int i=0; i<n; ++i) {
            if (vs[i]%n==0 || n%vs[i]==0) {
                swap(vs[i], vs[n-1]);
                ans += counts(n-1, vs);
                swap(vs[i], vs[n-1]);
            }
        }
        return ans;
    }
};
```

written by love_FDU_llp original link here

Solution 3

The trick is: Arrange the values starting from the end of the array.

```
public class Solution {
    public int countArrangement(int N) {
        dfs(N, N, new boolean[N + 1]);
        return count;
    }

    int count = 0;

    void dfs(int N, int k, boolean[] visited) {
        if (k == 0) {
            count++;
            return;
        }
        for (int i = 1; i <= N; i++) {
            if (visited[i] || k % i != 0 && i % k != 0) {
                continue;
            }
            visited[i] = true;
            dfs(N, k - 1, visited);
            visited[i] = false;
        }
    }
}
```

written by ibmtp380 original link here

From Leetcoder.

Minesweeper

Let's play the minesweeper game (Wikipedia, online game)!

You are given a 2D char matrix representing the game board. **'M'** represents an **unrevealed** mine, **'E'** represents an **unrevealed** empty square, **'B'** represents a **revealed** blank square that has no adjacent (above, below, left, right, and all 4 diagonals) mines, **digit** ('1' to '8') represents how many mines are adjacent to this **revealed** square, and finally **'X'** represents a **revealed** mine.

Now given the next click position (row and column indices) among all the **unrevealed** squares ('M' or 'E'), return the board after revealing this position according to the following rules:

1. If a mine ('M') is revealed, then the game is over - change it to **'X'**.
2. If an empty square ('E') with **no adjacent mines** is revealed, then change it to revealed blank ('B') and all of its adjacent **unrevealed** squares should be revealed recursively.
3. If an empty square ('E') with **at least one adjacent mine** is revealed, then change it to a digit ('1' to '8') representing the number of adjacent mines.
4. Return the board when no more squares will be revealed.

Example 1:

```
Input:

[['E', 'E', 'E', 'E', 'E'],
 ['E', 'E', 'M', 'E', 'E'],
 ['E', 'E', 'E', 'E', 'E'],
 ['E', 'E', 'E', 'E', 'E']]

Click : [3,0]

Output:

[['B', '1', 'E', '1', 'B'],
 ['B', '1', 'M', '1', 'B'],
 ['B', '1', '1', '1', 'B'],
 ['B', 'B', 'B', 'B', 'B']]

Explanation:
```

Example 2:

Input:

```
[['B', '1', 'E', '1', 'B'],
 ['B', '1', 'M', '1', 'B'],
 ['B', '1', '1', '1', 'B'],
 ['B', 'B', 'B', 'B', 'B']]

Click : [1,2]
```

Output:

```
[['B', '1', 'E', '1', 'B'],
 ['B', '1', 'X', '1', 'B'],
 ['B', '1', '1', '1', 'B'],
 ['B', 'B', 'B', 'B', 'B']]
```

Explanation:

Note:

1. The range of the input matrix's height and width is [1,50].
2. The click position will only be an unrevealed square ('M' or 'E'), which also means the input board contains at least one clickable square.
3. The input board won't be a stage when game is over (some mines have been revealed).
4. For simplicity, not mentioned rules should be ignored in this problem. For example, you **don't** need to reveal all the unrevealed mines when the game is over, consider any cases that you will win the game or flag any squares.

Solution 1

This is a typical `Search` problem, either by using `DFS` or `BFS`. Search rules:

1. If click on a mine ('`M`'), mark it as '`X`', stop further search.
2. If click on an empty cell ('`E`'), depends on how many surrounding mine:
   2.1 Has surrounding mine(s), mark it with number of surrounding mine(s), stop further search.
   2.2 No surrounding mine, mark it as '`B`', continue search its `8` neighbors.

DFS solution.

```
public class Solution {
    public char[][] updateBoard(char[][] board, int[] click) {
        int m = board.length, n = board[0].length;
        int row = click[0], col = click[1];

        if (board[row][col] == 'M') { // Mine
            board[row][col] = 'X';
        }
        else { // Empty
            // Get number of mines first.
            int count = 0;
            for (int i = -1; i < 2; i++) {
                for (int j = -1; j < 2; j++) {
                    if (i == 0 && j == 0) continue;
                    int r = row + i, c = col + j;
                    if (r < 0 || r >= m || c < 0 || c < 0 || c >= n) continue;
                    if (board[r][c] == 'M' || board[r][c] == 'X') count++;
                }
            }

            if (count > 0) { // If it is not a 'B', stop further DFS.
                board[row][col] = (char)(count + '0');
            }
            else { // Continue DFS to adjacent cells.
                board[row][col] = 'B';
                for (int i = -1; i < 2; i++) {
                    for (int j = -1; j < 2; j++) {
                        if (i == 0 && j == 0) continue;
                        int r = row + i, c = col + j;
                        if (r < 0 || r >= m || c < 0 || c < 0 || c >= n) continue
;
                        if (board[r][c] == 'E') updateBoard(board, new int[] {r,
c});
                    }
                }
            }
        }

        return board;
    }
}
```

BFS solution. As you can see the basic logic is almost the same as DFS. Only added a

queue to facilitate BFS.

```
public class Solution {
    public char[][] updateBoard(char[][] board, int[] click) {
        int m = board.length, n = board[0].length;
        Queue<int[]> queue = new LinkedList<>();
        queue.add(click);

        while (!queue.isEmpty()) {
            int[] cell = queue.poll();
            int row = cell[0], col = cell[1];

            if (board[row][col] == 'M') { // Mine
                board[row][col] = 'X';
            }
            else { // Empty
                // Get number of mines first.
                int count = 0;
                for (int i = -1; i < 2; i++) {
                    for (int j = -1; j < 2; j++) {
                        if (i == 0 && j == 0) continue;
                        int r = row + i, c = col + j;
                        if (r < 0 || r >= m || c < 0 || c < 0 || c >= n) continue
;

                        if (board[r][c] == 'M' || board[r][c] == 'X') count++;
                    }
                }

                if (count > 0) { // If it is not a 'B', stop further DFS.
                    board[row][col] = (char)(count + '0');
                }
                else { // Continue BFS to adjacent cells.
                    board[row][col] = 'B';
                    for (int i = -1; i < 2; i++) {
                        for (int j = -1; j < 2; j++) {
                            if (i == 0 && j == 0) continue;
                            int r = row + i, c = col + j;
                            if (r < 0 || r >= m || c < 0 || c < 0 || c >= n) cont
inue;

                            if (board[r][c] == 'E') {
                                queue.add(new int[] {r, c});
                                board[r][c] = 'B'; // Avoid to be added again.
                            }
                        }
                    }
                }
            }
        }

        return board;
    }
}
```

written by shawngao original link here

Solution 2

```
public class Solution {
    public char[][] updateBoard(char[][] board, int[] click) {
        int x = click[0], y = click[1];
        if (board[x][y] == 'M') {
            board[x][y] = 'X';
            return board;
        }

        dfs(board, x, y);
        return board;
    }

    int[] dx = {-1, 0, 1, -1, 1, 0, 1, -1};
    int[] dy = {-1, 1, 1, 0, -1, -1, 0, 1};
    private void dfs(char[][] board, int x, int y) {
        if (x < 0 || x >= board.length || y < 0 || y >= board[0].length || board[
x][y] != 'E')  return;

        int num = getNumsOfBombs(board, x, y);

        if (num == 0) {
            board[x][y] = 'B';
            for (int i = 0; i < 8; i++) {
                int nx = x + dx[i], ny = y + dy[i];
                dfs(board, nx, ny);
            }
        } else {
            board[x][y] = (char)('0' + num);
        }

    }

    private int getNumsOfBombs(char[][] board, int x, int y) {
        int num = 0;
        for (int i = -1; i <= 1; i++) {
            for (int j = -1; j <= 1; j++) {
                int nx = x + i, ny = y + j;
                if (nx < 0 || nx >= board.length || ny < 0 || ny >= board[0].leng
th)    continue;
                if (board[nx][ny] == 'M' || board[nx][ny] == 'X') {
                    num++;
                }
            }
        }
        return num;
    }
}
```

written by tankztc original link here

Solution 3

Make sure you know the rule of updating the board.

1. When clicking on the unrevealed mine, just update the current cell to 'X'.
2. When clicking on the cell with mines nearby (8 neighbors including diagonals), just update the count of mines in the neighborhood.
3. When clicking on the cell with 0 mine nearby, all the 8 neighbors also need to be checked and updated. And since once a cell is visited, its character will be changed for sure, the board will record the visited cells itself so no worry about revisiting.

ps: I know the format is somewhat stupid lol.

```java
public char[][] updateBoard(char[][] board, int[] click) {
    int x = click[0];
    int y = click[1];
    if(board[x][y] == 'M') board[x][y] = 'X';
    else if(countmines(board,x,y)>0) board[x][y] = (char)(countmines(board,x,y) +
'0');
    else update(board,x,y);
    return board;
}
private void update(char[][] board, int i, int j){
    if(i<0||i>=board.length||j<0||j>=board[0].length) return;
    if(board[i][j]=='E'){
        if(countmines(board,i,j)==0) {
            board[i][j] = 'B';
            update(board,i,j-1);
            update(board,i-1,j);
            update(board,i,j+1);
            update(board,i+1,j);
            update(board,i-1,j-1);
            update(board,i+1,j+1);
            update(board,i+1,j-1);
            update(board,i-1,j+1);
        }
        else{
            board[i][j] = (char)(countmines(board,i,j) + '0');
        }
    }
}

private int countmines(char[][] board,int i, int j){ // just count mines in the ne
ighborhood.
    int count = 0;
    if(i-1>=0&&board[i-1][j]=='M')count++;
    if(i+1<board.length&&board[i+1][j]=='M')count++;
    if(j-1>=0&&board[i][j-1]=='M')count++;
    if(j+1<board[0].length&&board[i][j+1]=='M')count++;
    if(i-1>=0&&j-1>=0&&board[i-1][j-1]=='M')count++;
    if(i+1<board.length&&j+1<board[0].length&&board[i+1][j+1]=='M') count++;
    if(i-1>=0&&j+1<board[0].length&&board[i-1][j+1]=='M') count++;
    if(i+1<board.length&&j-1>=0&&board[i+1][j-1]=='M') count++;
    return count;
}
```

written by Nakanu original link here

Minimum Absolute Difference in BST

Given a binary search tree with non-negative values, find the minimum absolute difference between values of any two nodes.

Example:

```
Input:

   1
    \
     3
    /
   2

Output:
1

Explanation:
The minimum absolute difference is 1, which is the difference between 2 and 1 (or b
etween 2 and 3).
```

Note: There are at least two nodes in this BST.

Solution 1

The most common idea is to first `inOrder` traverse the tree and compare the delta between each of the adjacent values. It's guaranteed to have the correct answer because it is a `BST` thus `inOrder` traversal values are `sorted`.

Solution 1 - In-Order traverse, time complexity O(N), space complexity O(1).

```
public class Solution {
    int min = Integer.MAX_VALUE;
    Integer prev = null;

    public int getMinimumDifference(TreeNode root) {
        if (root == null) return min;

        getMinimumDifference(root.left);

        if (prev != null) {
            min = Math.min(min, root.val - prev);
        }
        prev = root.val;

        getMinimumDifference(root.right);

        return min;
    }

}
```

What if it is not a `BST`? (Follow up of the problem) The idea is to put values in a TreeSet and then every time we can use `O(lgN)` time to lookup for the nearest values.

Solution 2 - Pre-Order traverse, time complexity O(NlgN), space complexity O(N).

```java
public class Solution {
    TreeSet<Integer> set = new TreeSet<>();
    int min = Integer.MAX_VALUE;

    public int getMinimumDifference(TreeNode root) {
        if (root == null) return min;

        if (!set.isEmpty()) {
            if (set.floor(root.val) != null) {
                min = Math.min(min, Math.abs(root.val - set.floor(root.val)));
            }
            if (set.ceiling(root.val) != null) {
                min = Math.min(min, Math.abs(root.val - set.ceiling(root.val)));
            }
        }

        set.add(root.val);

        getMinimumDifference(root.left);
        getMinimumDifference(root.right);

        return min;
    }
}
```

written by shawngao original link here

Solution 2

In-order traversal of BST yields sorted sequence. So, we just need to subtract the previous element from the current one, and keep track of the minimum. We need O(1) memory as we only store the previous element, but we still need O(h) for the stack.

```
void inorderTraverse(TreeNode* root, int& val, int& min_dif) {
    if (root->left != NULL) inorderTraverse(root->left, val, min_dif);
    if (val >= 0) min_dif = min(min_dif, root->val - val);
    val = root->val;
    if (root->right != NULL) inorderTraverse(root->right, val, min_dif);
}
int getMinimumDifference(TreeNode* root) {
    auto min_dif = INT_MAX, val = -1;
    inorderTraverse(root, val, min_dif);
    return min_dif;
}
```

Another solution with the member variables (6 lines):

```
class Solution {
    int min_dif = INT_MAX, val = -1;
public:
int getMinimumDifference(TreeNode* root) {
    if (root->left != NULL) getMinimumDifference(root->left);
    if (val >= 0) min_dif = min(min_dif, root->val - val);
    val = root->val;
    if (root->right != NULL) getMinimumDifference(root->right);
    return min_dif;
}
```

written by votrubac original link here

Solution 3

Since this is a BST, the inorder traversal of its nodes results in a sorted list of values. Thus, the minimum absolute difference must occur in any adjacently traversed nodes. I use the global variable "prev" to keep track of each node's inorder predecessor.

```
public class Solution {

    int minDiff = Integer.MAX_VALUE;
    TreeNode prev;

    public int getMinimumDifference(TreeNode root) {
        inorder(root);
        return minDiff;
    }

    public void inorder(TreeNode root) {
        if (root == null) return;
        inorder(root.left);
        if (prev != null) minDiff = Math.min(minDiff, root.val - prev.val);
        prev = root;
        inorder(root.right);
    }

}
```

written by compton_scatter original link here

From Leetcoder.

Lonely Pixel I

Given a picture consisting of black and white pixels, find the number of**black** lonely pixels.

The picture is represented by a 2D char array consisting of 'B' and 'W', which means black and white pixels respectively.

A black lonely pixel is character 'B' that located at a specific position where the same row and same column don't have any other black pixels.

Example:

```
Input:
[['W', 'W', 'B'],
 ['W', 'B', 'W'],
 ['B', 'W', 'W']]

Output: 3
Explanation: All the three 'B's are black lonely pixels.
```

Note:

1. The range of width and height of the input 2D array is [1,500].

Solution 1

O(nm) Time, O(n+m) Space Solution:

```
public int findLonelyPixel(char[][] picture) {
    int n = picture.length, m = picture[0].length;

    int[] rowCount = new int[n], colCount = new int[m];
    for (int i=0;i<n;i++)
        for (int j=0;j<m;j++)
            if (picture[i][j] == 'B') { rowCount[i]++; colCount[j]++; }

    int count = 0;
    for (int i=0;i<n;i++)
        for (int j=0;j<m;j++)
            if (picture[i][j] == 'B' && rowCount[i] == 1 && colCount[j] == 1) cou
nt++;

    return count;
}
```

O(nm) Time, O(1) Space Solution:

```
public int findLonelyPixel(char[][] picture) {
    int n = picture.length, m = picture[0].length;

    int firstRowCount = 0;
    for (int i=0;i<n;i++)
        for (int j=0;j<m;j++)
            if (picture[i][j] == 'B') {
                if (picture[0][j] < 'Y' && picture[0][j] != 'V') picture[0][j]++;
                if (i == 0) firstRowCount++;
                else if (picture[i][0] < 'Y' && picture[i][0] != 'V') picture[i][
0]++;
            }

    int count = 0;
    for (int i=0;i<n;i++)
        for (int j=0;j<m;j++)
            if (picture[i][j] < 'W' && (picture[0][j] == 'C' || picture[0][j] ==
'X')) {
                if (i == 0) count += firstRowCount == 1 ? 1 : 0;
                else if (picture[i][0] == 'C' || picture[i][0] == 'X') count++;
            }

    return count;
}
```

written by compton_scatter original link here

Solution 2

thought is very simple, we can easily count how many times B occurs in each row. But how can we know if this col has existing B?
for example, input is
W B B B
B W W W
W W W B
W W W B
we can maintain an array calls colArray[], which is used to record how many times the B occurs in each column. Then solution is simple

```
public class Solution {
    public int findLonelyPixel(char[][] picture) {
        if (picture == null || picture.length == 0 || picture[0].length == 0) ret
urn 0;

        int[] colArray = new int[picture[0].length];
        for (int i = 0; i < picture.length; i++) {
            for (int j = 0; j < picture[0].length; j++) {
                if (picture[i][j] == 'B') colArray[j]++;
            }
        }

        int ret = 0;
        for (int i = 0; i < picture.length; i++) {
            int count = 0, pos = 0;
            for (int j = 0; j < picture[0].length; j++) {
                if (picture[i][j] == 'B') {
                    count++;
                    pos = j;
                }
            }
            if (count == 1 && colArray[pos] == 1) ret++;
        }
        return ret;
    }
}
```

written by lusoul original link here

Solution 3

C++

```
/**
 * suppose matrix is m*n, there is at most min(m, n) lonely pixels, because there
could be no more than 1 in each row, or column;
 * therefore, if we record num of black pixel on each row and column, we can easil
y tell whether each pixel is lonely or NO.
 *     _0_1_2_
 *  0 | 0 0 1   rows[0] = 1
 *  1 | 0 1 0   rows[1] = 1
 *  2 | 1 0 0   rows[2] = 1
 *
 * cols[0][1][2]
 *     1  1  1
 */
class Solution {
public:
    int findLonelyPixel(vector<vector<char>>& pic) {
        int m = pic.size();
        int n = pic[0].size();
        vector<int> rows = vector<int>(m, 0);
        vector<int> cols = vector<int>(n, 0);

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                rows[i] += pic[i][j] == 'B';
                cols[j] += pic[i][j] == 'B';
            }
        }

        int lonely = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n && rows[i] > 0; j++) {
                lonely += pic[i][j] == 'B' && rows[i] == 1 && cols[j] == 1;
            }
        }

        return lonely;
    }
};
```

Java

```java
public class Solution {
    public int findLonelyPixel(char[][] pic) {
        int m = pic.length;
        int n = pic[0].length;
        int[] rows = new int[m];
        int[] cols = new int[n];

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                rows[i] += pic[i][j] == 'B' ? 1 : 0;
                cols[j] += pic[i][j] == 'B' ? 1 : 0;
            }
        }

        int lonely = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n && rows[i] > 0; j++) {
                lonely += (pic[i][j] == 'B' && rows[i] == 1 && cols[j] == 1) ? 1
: 0;
            }
        }

        return lonely;
    }
}
```

written by alexander original link here

From Leetcoder.

K-diff Pairs in an Array

Given an array of integers and an integer **k**, you need to find the number of **unique** k-diff pairs in the array. Here a **k-diff** pair is defined as an integer pair (i, j), where **i** and **j** are both numbers in the array and their absolute difference is **k**.

Example 1:

```
Input: [3, 1, 4, 1, 5], k = 2
Output: 2
Explanation: There are two 2-diff pairs in the array, (1, 3) and (3, 5).Although we
 have two 1s in the input, we should only return the number of unique pairs.
```

Example 2:

```
Input:[1, 2, 3, 4, 5], k = 1
Output: 4
Explanation: There are four 1-diff pairs in the array, (1, 2), (2, 3), (3, 4) and (
4, 5).
```

Example 3:

```
Input: [1, 3, 1, 5, 4], k = 0
Output: 1
Explanation: There is one 0-diff pair in the array, (1, 1).
```

Note:

1. The pairs (i, j) and (j, i) count as the same pair.
2. The length of the array won't exceed 10,000.
3. All the integers in the given input belong to the range: [-1e7, 1e7].

Solution 1

```
public class Solution {
    public int findPairs(int[] nums, int k) {
        if (nums == null || nums.length == 0 || k < 0)   return 0;

        Map<Integer, Integer> map = new HashMap<>();
        int count = 0;
        for (int i : nums) {
            map.put(i, map.getOrDefault(i, 0) + 1);
        }

        for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
            if (k == 0) {
                //count how many elements in the array that appear more than twic
e.
                if (entry.getValue() >= 2) {
                    count++;
                }
            } else {
                if (map.containsKey(entry.getKey() + k)) {
                    count++;
                }
            }
        }

        return count;
    }
}
```

written by tankztc original link here

Solution 2

The problem is just a variant of 2-sum.
Update: Fixed a bug that can cause integer subtraction overflow.

```
public int findPairs(int[] nums, int k) {
    int ans = 0;
    Arrays.sort(nums);
    for (int i = 0, j = 0; i < nums.length; i++) {
        for (j = Math.max(j, i + 1); j < nums.length && (long) nums[j] - nums[i]
< k; j++) ;
        if (j < nums.length && (long) nums[j] - nums[i] == k) ans++;
        while (i + 1 < nums.length && nums[i] == nums[i + 1]) i++;
    }
    return ans;
}
```

written by lixx2100 original link here

Solution 3

```
    def findPairs(self, nums, k):
        return len(set(nums)&{n+k for n in nums}) if k>0 else sum(v>1 for v in col
lections.Counter(nums).values()) if k==0 else 0
```

which is equivalent to:

```
    def findPairs(self, nums, k):
        if k>0:
            return len(set(nums)&set(n+k for n in nums))
        elif k==0:
            sum(v>1 for v in collections.Counter(nums).values())
        else:
            return 0
```

written by o_sharp original link here

From Leetcoder.

Lonely Pixel II

Given a picture consisting of black and white pixels, and a positive integer N, find the number of black pixels located at some specific row **R** and column **C** that align with all the following rules:

1. Row R and column C both contain exactly N black pixels.
2. For all rows that have a black pixel at column C, they should be exactly the same as row R

The picture is represented by a 2D char array consisting of 'B' and 'W', which means black and white pixels respectively.

Example:

```
Input:
[['W', 'B', 'W', 'B', 'B', 'W'],
 ['W', 'B', 'W', 'B', 'B', 'W'],
 ['W', 'B', 'W', 'B', 'B', 'W'],
 ['W', 'W', 'B', 'W', 'B', 'W']]

N = 3
Output: 6
Explanation: All the bold 'B' are the black pixels we need (all 'B's at column 1 an
d 3).
        0    1    2    3    4    5         column index

0    [['W', 'B', 'W', 'B', 'B', 'W'],
1     ['W', 'B', 'W', 'B', 'B', 'W'],
2     ['W', 'B', 'W', 'B', 'B', 'W'],
3     ['W', 'W', 'B', 'W', 'B', 'W']]
row index

Take 'B' at row R = 0 and column C = 1 as an example:
Rule 1, row R = 0 and column C = 1 both have exactly N = 3 black pixels.
Rule 2, the rows have black pixel at column C = 1 are row 0, row 1 and row 2. They
are exactly the same as row R = 0.
```

Note:

1. The range of width and height of the input 2D array is [1,200].

Solution 1

The difficult parts are validating the two rules:

1. Row `R` and column `C` both contain exactly `N` black pixels.
2. For all rows that have a black pixel at column `C`, they should be exactly the same as row `R`

My solution:

1. Scan each row. If that row has `N` black pixels, put a string `signature`, which is concatenation of each characters in that row, as key and how many times we see that `signature` into a HashMap. Also during scan each row, we record how many black pixels in each column in an array `colCount` and will use it in step 2.
   For input:
   [['W', 'B', 'W', 'B', 'B', 'W'],
   ['W', 'B', 'W', 'B', 'B', 'W'],
   ['W', 'B', 'W', 'B', 'B', 'W'],
   ['W', 'W', 'B', 'W', 'B', 'B']]
   We will get a HashMap:
   {"WBWBBW": 3, "WWBWBB": 1}
   and colCount array:
   [0, 3, 1, 3, 4, 1]
2. Go through the HashMap and if the count of one `signature` is `N`, those rows potentially contain black pixels we are looking for. Then we validate each of those columns. For each column of them has `N` black pixels (lookup in `colCount` array), we get `N` valid black pixels.
   For above example, only the first `signature` "WBWBBW" has count == 3. We validate 3 column 1, 3, 4 where char == 'B', and column 1 and 3 have 3 'B', then answer is 2 * 3 = 6.

Time complexity analysis:
Because we only scan the matrix for one time, time complexity is O(m*n). m = number of rows, n = number of columns.

```java
public class Solution {
    public int findBlackPixel(char[][] picture, int N) {
        int m = picture.length;
        if (m == 0) return 0;
        int n = picture[0].length;
        if (n == 0) return 0;

        Map<String, Integer> map = new HashMap<>();
        int[] colCount = new int[n];

        for (int i = 0; i < m; i++) {
            String key = scanRow(picture, i, N, colCount);
            if (key.length() != 0) {
                map.put(key, map.getOrDefault(key, 0) + 1);
            }
        }

        int result = 0;
        for (String key : map.keySet()) {
            if (map.get(key) == N) {
                for (int j = 0; j < n; j++) {
                    if (key.charAt(j) == 'B' && colCount[j] == N) {
                        result += N;
                    }
                }
            }
        }

        return result;
    }

    private String scanRow(char[][] picture, int row, int target, int[] colCount)
{
        int n = picture[0].length;
        int rowCount = 0;
        StringBuilder sb = new StringBuilder();

        for (int j = 0; j < n; j++) {
            if (picture[row][j] == 'B') {
                rowCount++;
                colCount[j]++;
            }
            sb.append(picture[row][j]);
        }

        if (rowCount == target) return sb.toString();
        return "";
    }
}
```

written by shawngao original link here

Solution 2

Mainly I group and count equal rows. Look for rows that appear N times and that have N black pixels. If you find one, add N for each of its black columns that doesn't have extra black pixels (in other rows).

```
def findBlackPixel(self, picture, N):
    ctr = collections.Counter(map(tuple, picture))
    cols = [col.count('B') for col in zip(*picture)]
    return sum(N * zip(row, cols).count(('B', N))
               for row, count in ctr.items()
               if count == N == row.count('B'))
```

written by StefanPochmann original link here

Solution 3

Analysis

```
/**
 * Steps:
 * >> 1. create map<int, set<int>> cols, rows; -- to store black dots on that row;
 *
 *     _0_1_2_3_4_5_
 *  0 | 0 l 0 1 1 0     rows[0] = {1, 3, 4}
 *  1 | 0 l 0 1 1 0     rows[1] = {1, 3, 4}
 *  2 | 0 l 0 1 1 0     rows[2] = {1, 3, 4}
 *  3 | 0 0 1 0 1 0     rows[3] = {  2,  4}
 *
 * >> 2. for every pixel meet rule 1, that is: pic[i][j] == 'B' && rows[i].size()
== N && cols[j].size() == N
 *       check rule2: for every row k in cols[j];  check that row[k] = row[i];
 *
 * We can tell the 6 black pixel in col 1 and col 3 are lonely pixels
 *     _0_1_2_3_4_5_
 *  0 | 0 L 0 L 1 0     rows[0] = {1, 3, 4}  =
 *  1 | 0 L 0 L 1 0     rows[1] = {1, 3, 4}  =
 *  2 | 0 L 0 L 1 0     rows[2] = {1, 3, 4}
 *  3 | 0 0 1 0 1 0     rows[3] = {  2,  4}
 *
 */
```

C++

```
class Solution {
public:
    int findBlackPixel(vector<vector<char>>& pic, int N) {
        int m = pic.size();
        int n = pic[0].size();
        unordered_map<int, set<int>> rows;  // black pixels in each row
        unordered_map<int, set<int>> cols;  // black pixels in each col
        /* 1. create map<int, set<int>> cols, rows; -- to store black dots on tha
t row; */
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (pic[i][j] == 'B') {
                    rows[i].insert(j);
                    cols[j].insert(i);
                }
            }
        }
        /* 2. for every pixel meet rule 1: pic[i][j] == 'B' && rows[i].size() == N
&& cols[j].size() == N */
        int lonelys = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n && rows.count(i); j++) {
                if (pic[i][j] == 'B' && rows[i].size() == N && cols[j].size() ==
N) {   // rule 1 fulfilled
                    /* check rule2: for every row k in cols[j];  check that row[k
] = row[i]; */
                    bool lonely = true;
                    for (int r : cols[j]) {
                        if (rows[r] != rows[i]) {
                            lonely = false; break;
                        }
                    }
                    lonelys += lonely;
                }
            }
        }
        return lonelys;
    }
};
```

Java

```java
public class Solution {
    public int findBlackPixel(char[][] pic, int N) {
        int m = pic.length;
        int n = pic[0].length;
        Map<Integer, Set<Integer>> rows = new HashMap<Integer, Set<Integer>>(); /
/ black pixels in each row;
        Map<Integer, Set<Integer>> cols = new HashMap<Integer, Set<Integer>>(); /
/ black pixels in each col;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (pic[i][j] == 'B') {
                    if (!rows.containsKey(i)) { rows.put(i, new HashSet<Integer>(
)); }
                    if (!cols.containsKey(j)) { cols.put(j, new HashSet<Integer>(
)); }
                    rows.get(i).add(j);
                    cols.get(j).add(i);
                }
            }
        }
        int lonelys = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n && rows.containsKey(i); j++) {
                if (pic[i][j] == 'B' && rows.get(i).size() == N && cols.containsK
ey(j) && cols.get(j).size() == N) {   // rule 1 fulfilled
                    int lonely = 1;
                    for (int row : cols.get(j)) {
                        if (!rows.get(i).equals(rows.get(row))) {
                            lonely = 0; break;
                        }
                    }
                    lonelys += lonely;
                }
            }
        }
        return lonelys;
    }
}
```

written by alexander original link here

From Leetcoder.

Encode and Decode TinyURL

> Note: This is a companion problem to the System Design problem: Design TinyURL.

TinyURL is a URL shortening service where you enter a URL such as `https://leetcode.com/problems/design-tinyurl` and it returns a short URL such as `http://tinyurl.com/4e9iAk`.

Design the `encode` and `decode` methods for the TinyURL service. There is no restriction on how your encode/decode algorithm should work. You just need to ensure that a URL can be encoded to a tiny URL and the tiny URL can be decoded to the original URL.

Solution 1

My first solution produces short URLs like `http://tinyurl.com/0`, `http://tinyurl.com/1`, etc, in that order.

```
class Codec:

    def __init__(self):
        self.urls = []

    def encode(self, longUrl):
        self.urls.append(longUrl)
        return 'http://tinyurl.com/' + str(len(self.urls) - 1)

    def decode(self, shortUrl):
        return self.urls[int(shortUrl.split('/')[-1])]
```

Using increasing numbers as codes like that is simple but has some disadvantages, which the below solution fixes:

- If I'm asked to encode the same long URL several times, it will get several entries. That wastes codes and memory.
- People can find out how many URLs have already been encoded. Not sure I want them to know.
- People might try to get special numbers by spamming me with repeated requests shortly before their desired number comes up.
- Only using digits means the codes can grow unnecessarily large. Only offers a million codes with length 6 (or smaller). Using six digits or lower or upper case letters would offer $(10+26*2)^6$ = 56,800,235,584 codes with length 6.

The following solution doesn't have these problems. It produces short URLs like `http://tinyurl.com/KtLa2U`, using a random code of six digits or letters. If a long URL is already known, the existing short URL is used and no new entry is generated.

```
class Codec:

    alphabet = string.ascii_letters + '0123456789'

    def __init__(self):
        self.url2code = {}
        self.code2url = {}

    def encode(self, longUrl):
        while longUrl not in self.url2code:
            code = ''.join(random.choice(Codec.alphabet) for _ in range(6))
            if code not in self.code2url:
                self.code2url[code] = longUrl
                self.url2code[longUrl] = code
        return 'http://tinyurl.com/' + self.url2code[longUrl]

    def decode(self, shortUrl):
        return self.code2url[shortUrl[-6:]]
```

It's possible that a randomly generated code has already been generated before. In that case, another random code is generated instead. Repeat until we have a code that's not already in use. How long can this take? Well, even if we get up to using half of the code space, which is a whopping $62^6/2$ = 28,400,117,792 entries, then each code has a 50% chance of not having appeared yet. So the expected/average number of attempts is 2, and for example only one in a billion URLs takes more than 30 attempts. And if we ever get to an even larger number of entries and this does become a problem, then we can just use length 7. We'd need to anyway, as we'd be running out of available codes.

written by StefanPochmann original link here

Solution 2

```
class Solution {
public:

    // Encodes a URL to a shortened URL.
    string encode(string longUrl) {
        return longUrl;
    }

    // Decodes a shortened URL to its original URL.
    string decode(string shortUrl) {
        return shortUrl;
    }
};
```

written by xplorld original link here

Solution 3

below is the tiny url solution in java, also this is the similar method in industry. In industry, most of shorten url service is by database, one auto increasing long number as primary key. whenever a long url need to be shorten, append to the database, and return the primary key number. (the database is very easy to distribute to multiple machine like HBase, or even you can use the raw file system to store data and improve performance by shard and replica).
Note, it's meaningless to promise the same long url to be shorten as the same short url. if you do the promise and use something like hash to check existing, the benefit is must less than the cost.
Note: if you want the shorted url contains '0-9a-zA-Z' instead of '0-9', then you need to use 62 number system, not 10 number system(decimal) to convert the primary key number. like 123->'123' in decimal, 123->'1Z' in 62 number system (or '0001Z' for align).

```
public class Codec {
    List<String> urls = new ArrayList<String>();
    // Encodes a URL to a shortened URL.
    public String encode(String longUrl) {
        urls.add(longUrl);
        return String.valueOf(urls.size()-1);
    }

    // Decodes a shortened URL to its original URL.
    public String decode(String shortUrl) {
        int index = Integer.valueOf(shortUrl);
        return (index<urls.size())?urls.get(index):"";
    }
}
```

written by qiu5 original link here

From Leetcoder.

Construct Binary Tree from String

You need to construct a binary tree from a string consisting of parenthesis and integers.

The whole input represents a binary tree. It contains an integer followed by zero, one or two pairs of parenthesis. The integer represents the root's value and a pair of parenthesis contains a child binary tree with the same structure.

You always start to construct the **left** child node of the parent first if it exists.

Example:

```
Input: "4(2(3)(1))(6(5))"
Output: return the tree root node representing the following tree:

       4
     /   \
    2     6
   / \   /
  3   1 5
```

Note:

1. There will only be `'('`, `')'`, `'-'` and `'0'` ~ `'9'` in the input string.
2. An empty tree is represented by `""` instead of `"()"`.

Solution 1

I change a string like `'4(2(3)(1))(6(5))'` to `'t(4,t(2,t(3),t(1)),t(6,t(5)))'` and then just let Python evaluate that (with the help of my TreeNode constructor).

```
def str2tree(self, s):
    def t(val, left=None, right=None):
        node, node.left, node.right = TreeNode(val), left, right
        return node
    return eval('t(' + s.replace('(', ',t(') + ')') if s else None
```

written by StefanPochmann original link here

Solution 2

This is a pretty common approach with static index variable.

```
let i = 0;

var construct = function (s) {
    let numStr = [];

    while (s[i] === '-' || s[i] >= '0' && s[i] <= '9')
        numStr.push(s[i++]);

    if (numStr.length === 0)
        return null;

    let node = new TreeNode(Number(numStr.join('')));
    if (s[i] === '(')
        i++, node.left = construct(s), i++;
    if (s[i] === '(')
        i++, node.right = construct(s), i++;
    return node;
}

var str2tree = function(s) {
    i = 0;
    return construct(s);
};
```

written by dettier original link here

Solution 3

We perform a recursive solution. There are four cases for what the string might look like:

1. empty
2. [integer]
3. [integer] ([tree])
4. [integer] ([tree]) ([tree])

When there is no '(', we are in one of the first two cases and proceed appropriately. Else, we find the index "jx" of the ')' character that marks the end of the first tree. We do this by keeping a tally of how many left brackets minus right brackets we've seen. When we've seen 0, we must be at the end of the first tree. The second tree is going to be the expression S[jx + 2: -1], which might be empty if we are in case #3.

```
def str2tree(self, S):
    ix = S.find('(')
    if ix < 0:
        return TreeNode(int(S)) if S else None

    bal = 0
    for jx, u in enumerate(S):
        if u == '(': bal += 1
        if u == ')': bal -= 1
        if jx > ix and bal == 0:
            break

    root = TreeNode(int(S[:ix]))
    root.left = self.str2tree(S[ix+1:jx])
    root.right = self.str2tree(S[jx+2:-1])
    return root
```

written by awice original link here

From Leetcoder.

Minimum Time Difference

Given a list of 24-hour clock time points in "Hour:Minutes" format, find the minimum **minutes** difference between any two time points in the list.

Example 1:

```
Input: ["23:59","00:00"]
Output: 1
```

Note:

1. The number of time points in the given list is at least 2 and won't exceed 20000.
2. The input time is legal and ranges from 00:00 to 23:59.

Solution 1

There is only 24 * 60 = 1440 possible time points. Just create a boolean array, each element stands for if we see that time point or not. Then things become simple...

```
public class Solution {
    public int findMinDifference(List<String> timePoints) {
        boolean[] mark = new boolean[24 * 60];
        for (String time : timePoints) {
            String[] t = time.split(":");
            int h = Integer.parseInt(t[0]);
            int m = Integer.parseInt(t[1]);
            if (mark[h * 60 + m]) return 0;
            mark[h * 60 + m] = true;
        }

        int prev = 0, min = Integer.MAX_VALUE;
        int first = Integer.MAX_VALUE, last = Integer.MIN_VALUE;
        for (int i = 0; i < 24 * 60; i++) {
            if (mark[i]) {
                if (first != Integer.MAX_VALUE) {
                    min = Math.min(min, i - prev);
                }
                first = Math.min(first, i);
                last = Math.max(last, i);
                prev = i;
            }
        }

        min = Math.min(min, (24 * 60 - last + first));

        return min;
    }
}
```

written by shawngao original link here

Solution 2

O(nlog(n)) Time O(1) Space:

```
public int findMinDifference(List<String> timePoints) {
    Collections.sort(timePoints);
    int minDiff = Integer.MAX_VALUE;
    String prev = timePoints.get(timePoints.size()-1);
    for (String s : timePoints) {
        int prevMins = Integer.parseInt(prev.split(":")[0])*60 + Integer.parseInt
(prev.split(":")[1]);
        int curMins = Integer.parseInt(s.split(":")[0])*60 + Integer.parseInt(s.s
plit(":")[1]);
        int diff = curMins - prevMins;
        if (diff < 0) diff += 1440;
        minDiff = Math.min(minDiff, Math.min(diff, 1440 - diff));
        prev = s;
    }
    return minDiff;
}
```

O(n) Time O(1) Space. Note that, more accurately, this is O(1) time as the number of iterations of the first loop is limited to 1440 due to the pigeonhole principle.

```
public int findMinDifference(List<String> timePoints) {

    boolean[] timeSeen = new boolean[1440];
    for (String s : timePoints) {
        int mins = Integer.parseInt(s.split(":")[0])*60 + Integer.parseInt(s.spli
t(":")[1]);
        if (timeSeen[mins]) return 0;
        timeSeen[mins] = true;
    }

    Integer firstTimeSeen = null, prevTimeSeen = null, minDiff = Integer.MAX_VALU
E;
    for (int i=0;i<1440;i++) {
        if (!timeSeen[i]) continue;
        if (firstTimeSeen == null) {firstTimeSeen = i; prevTimeSeen = i;}
        else {
          minDiff = Math.min(minDiff, Math.min(i - prevTimeSeen, 1440 - i + prevT
imeSeen));
          prevTimeSeen = i;
        }
    }

    minDiff = Math.min(minDiff, Math.min(prevTimeSeen - firstTimeSeen, 1440 - pre
vTimeSeen + firstTimeSeen));
    return minDiff;
}
```

written by compton_scatter original link here

Solution 3

```
public class Solution {
    public int findMinDifference(List<String> timePoints) {
        int n = timePoints.size();
        List<Time> times = new ArrayList<>();
        for (String tp : timePoints) {
            String[] strs = tp.split(":");
            times.add(new Time(Integer.parseInt(strs[0]), Integer.parseInt(strs[1
])));
        }
        Collections.sort(times);
        Time earlist = times.get(0);
        times.add(new Time(earlist.h + 24, earlist.m));
        int minDiff = Integer.MAX_VALUE;
        for (int i = 0; i < n; i++) {
            int diff = (int) Math.abs(times.get(i).getDiff(times.get(i + 1)));
            minDiff = Math.min(minDiff, diff);
        }
        return minDiff;
    }

}

class Time implements Comparable<Time> {
    int h;
    int m;
    public Time(int h, int m) {
        this.h = h;
        this.m = m;
    }

    public int compareTo(Time other) {
        if (this.h == other.h) {
            return this.m - other.m;
        }
        return this.h - other.h;
    }

    public int getDiff(Time other) {
        return (this.h - other.h) * 60 + (this.m - other.m);
    }
}
```

written by zzwcsong original link here

From Leetcoder.

Reverse String II

Given a string and an integer k, you need to reverse the first k characters for every 2k characters counting from the start of the string. If there are less than k characters left, reverse all of them. If there are less than 2k but greater than or equal to k characters, then reverse the first k characters and left the other as original.

Example:

```
Input: s = "abcdefg", k = 2
Output: "bacdfeg"
```

Restrictions:

1. The string consists of lower English letters only.
2. Length of the given string and k will in the range [1, 10000]

Solution 1

For every block of 2k characters starting with position i, we want to replace S[i:i+k] with it's reverse.

```
def reverseStr(self, s, k):
    s = list(s)
    for i in xrange(0, len(s), 2*k):
        s[i:i+k] = reversed(s[i:i+k])
    return "".join(s)
```

written by awice original link here

Solution 2

```java
public class Solution {
    public String reverseStr(String s, int k) {
        char[] arr = s.toCharArray();
        int n = arr.length;
        int i = 0;
        while(i < n) {
            int j = Math.min(i + k - 1, n - 1);
            swap(arr, i, j);
            i += 2 * k;
        }
        return String.valueOf(arr);
    }
    private void swap(char[] arr, int l, int r) {
        while (l < r) {
            char temp = arr[l];
            arr[l++] = arr[r];
            arr[r--] = temp;
        }
    }
}
```

written by sheva original link here

Solution 3

Tried to maximize usage of `StringBuilder` :)

```
public class Solution {
    public String reverseStr(String s, int k) {
        StringBuilder sb = new StringBuilder();

        int i = 0, j = 0;
        while (i < s.length()) {
            // first k
            j = i + k <= s.length() ? i + k : s.length();
            sb.append((new StringBuilder(s.substring(i, j))).reverse().toString()
);

            if (j >= s.length()) break;

            // second k
            i = j;
            j = i + k <= s.length() ? i + k : s.length();
            sb.append(s.substring(i, j));

            i = j;
        }

        return sb.toString();
    }

}
```

written by shawngao original link here

From Leetcoder.