1

```
// Forward declaration of the knows API.
bool knows(int a, int b);

class Solution {
public:
    int findCelebrity(int n) {
        int l = 0, r = n - 1;
        while (l < r) {
            if (knows(l, r)) ++l;
            else --r;
        }
        for (int i = 0; i < n; ++i) if (i != l) {
            if (knows(l, i) || !knows(i, l)) return -1;
        }
        return l;
    }
};
```

2.

```
int getInfluencer(vector<vector<bool> > M) {
        int cand=0;
        for(int i=1; i<M.size(); i++)
        {
                if(M[cand][i] == 1 || M[i][cand]==0)
                {
                        cand = i;
                }
        }
        // now verify cand is indeed an influencer
        for(int j=0; j<M.size(); j++)
        {
                if(j==cand) continue;
                if(M[cand][j]==1 || M[j][cand]==0) return -1;
        }
        return cand;
}
```

3.

```
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 itself
        // 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;
}
```

4. Consider a data structure composed of a hashtable H and an array A. The hashtable keys are the elements in the data structure and the values are their position in the array

1. insert(value): append the value to array and let i be its index in A. Set H[value]=i.
2. remove(value): We are going to replace the cell that contains value in A with the last element in A. let d be the last element in the array A at index m. let i be H[value], the index in the array of the value to be removed. Set A[i]=d, H[d]=i, decrease the size of the array by one, and remove value from H.
3. contains(value): return H.contains(value)
4. getRandomElement(): let r=random(current size of A). return A[r].

since the array needs to auto-increase in size, it's going to be amortize O(1) to add an element, but I guess that's OK.

5.

```
Node FlipTree ( Node root )
{
    if (root == NULL)
        return NULL;

   if( root.Left == NULL && root.Right == NULL)
    {
        return root;
    }

    Node newRoot = FlipTree(root.Left);

    root.Left.Left = root.Right;
    root.Left.Right = root;
    root.Left = NULL;
    root.Right = NULL;

    return newRoot;
}
```

6.

```
  int rob(vector<int> &num) {
    if(num.size()==0) return 0;
    vector<int> dp(num.size(), 0);
    dp[0]=num[0]; dp[1]=max(num[0],num[1]);
    for(int i=2;i<num.size();++i){
      dp[i]=max(num[i]+dp[i-2], dp[i-1]);
    }
    return dp[num.size()-1];
  }
```

7.

```
int p=0, q=0, n=A.size(), cur=0, ret=n+1;
while(p<n && q<n){
    cur += A[q];
    if(cur<s) q++;
    else{
        while(cur>=s){
            ret = min(ret, q-p+1);
            cur -= A[p];
            p++;
        }
        q++;
    }
}
return ret==n+1?0:ret;
```

O(NLogN) - search if a window of size k exists that satisfy the condition

```
public class Solution {
    public int minSubArrayLen(int s, int[] nums) {
        int i = 1, j = nums.length, min = 0;
        while (i <= j) {
            int mid = (i + j) / 2;
            if (windowExist(mid, nums, s)) {
                j = mid - 1;
                min = mid;
            } else i = mid + 1;
        }
        return min;
    }

    private boolean windowExist(int size, int[] nums, int s) {
        int sum = 0;
        for (int i = 0; i < nums.length; i++) {
            if (i >= size) sum -= nums[i - size];
            sum += nums[i];
            if (sum >= s) return true;
        }
        return false;
    }
}
```

Another O(NLogN) solution that first calculate cumulative sum and then for each starting point binary search for end position. This uses O(N) space

```
public class Solution {
 public int minSubArrayLen(int s, int[] nums) {
        int sum = 0, min = Integer.MAX_VALUE;

        int[] sums = new int[nums.length];
        for (int i = 0; i < nums.length; i++)
            sums[i] = nums[i] + (i == 0 ? 0 : sums[i - 1]);

        for (int i = 0; i < nums.length; i++) {
            int j = findWindowEnd(i, sums, s);
            if (j == nums.length) break;
            min = Math.min(j - i + 1, min);
        }

        return min == Integer.MAX_VALUE ? 0 : min;
    }

    private int findWindowEnd(int start, int[] sums, int s) {
        int i = start, j = sums.length - 1, offset = start == 0 ? 0 : sums[start - 1];
        while (i <= j) {
            int m = (i + j) / 2;
            int sum = sums[m] - offset;
        if (sum >= s) j = m - 1;
        else i = m + 1;
    }
    return i;
}
```

8.

```
    void solve(vector<int> &A, int begin, vector<vector<int>>& result){
        if(begin>=A.size()){//A.size()
            result.push_back(A);
            return;
        }

        for(int i=begin; i<A.size();++i){//i start from begin
            swap(A[begin], A[i]);
            solve(A, begin+1, result);//begin+1
            swap(A[begin], A[i]);
        }
    }

    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int> > result;
        solve(nums, 0, result);
        return result;
    }
```

9.

```
int jump(vector<int>& A) {
    int sc = 0; //minimum steps for reaching e
    int e = 0; //longest distance in current minimum step
    int m = 0;
    for(int i=0; i<A.size()-1; i++) {
        m = max(m, i+A[i]);
        if( i == e ) {
            sc++;
            e = m;
        }
    }
    return sc;
}
```

10.

```
 vector<int> findSubstring(string s, vector<string>& words) {
        unordered_map<string, int> counts;
        for (string word : words)
            counts[word]++;
        int n = s.length(), num = words.size(), len = words[0].length();
        vector<int> indexes;
        for (int i = 0; i < n - num * len + 1; i++) {
            unordered_map<string, int> seen;
            int j = 0;
            for (; j < num; j++) {
                string word = s.substr(i + j * len, len);
                if (counts.find(word) != counts.end()) {
                    seen[word]++;
                    if (seen[word] > counts[word])
                        break;
                }
                else break;
            }
            if (j == num) indexes.push_back(i);
        }
        return indexes;
    }
```

```
    // 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;
    }
```