# 数学(Mathematics)

## 离散数学(Discrete Mathematics)

### 图论(Graph Theory)

#### 图的遍历(Graph Traversal): DFS, BFS

|  |
| --- |
| 399. 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.  C++1 (3ms)  class Solution {  public:  double search(string left, string right, unordered\_map<string, unordered\_map<string, double>>& graph, unordered\_set<string>& set)  {  if(graph[left].find(right)!=graph[left].end()) return graph[left][right]; //find it in graph  for(auto g : graph[left])  {  if(set.find(g.first) == set.end()) //not visited yet  {  set.insert(g.first); //mark visit  double temp = search(g.first, right, graph, set);  if(temp>0) return temp\*g.second;  }  }  return 0;  }  vector<double> calcEquation(vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries) {  unordered\_map<string, unordered\_map<string, double>> graph;  vector<double> ans;  for(int i=0; i<values.size(); ++i)  {  graph[equations[i].first].insert(make\_pair(equations[i].second, values[i])); //create graphs  graph[equations[i].second].insert(make\_pair(equations[i].first, 1.0/values[i])); // values are all positive  }    for(auto q : queries)  {  unordered\_set<string> set;  double temp = search(q.first, q.second, graph, set);  ans.push\_back((temp>0) ? temp : -1.0);  }  return ans;  }  }; |
|  |
|  |
|  |

#### 最小生成树(Minimum Spanning Tree): Prim, Kruskal

#### 最短路径(Shortest Path): Dijkstra, Floyd

#### 最长路径(longest Path)

|  |
| --- |
| 368. Largest Divisible Subset  Given a set of **distinct** positive integers, find the largest subset such that every pair (Si, Sj) of elements in this subset satisfies: Si % Sj = 0 or Sj % Si = 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]  C++1  vector<int> largestDivisibleSubset(vector<int>& nums) {  sort(nums.begin(), nums.end());  int n=nums.size(), start=0, maxSize=0;  vector<int> link(n), linkLen(n), res;  for(int i=n-1; i>=0; --i)  {  for(int j=i; j<n; ++j)  {  if(nums[j]%nums[i]==0 && linkLen[i] < 1+linkLen[j])  {  linkLen[i] = 1+linkLen[j];  link[i] = j;  }  }  if(linkLen[i] > maxSize)  {  maxSize = linkLen[i];  start = i;  }  }  for(int i=0; i<maxSize; ++i)  {  res.push\_back(nums[start]);  start=link[start];  }  /\*  vector<int> res(maxSize);  for(int i=0; i<maxSize; ++i)  {  res[i] = nums[start];  start = link[divi];  }  \*/  return res;  } |
|  |
|  |

#### 传递闭包(Transitive Closure)

#### 关节点(Articulation Point - UndiGraph)

#### 拓扑排序(Topological Sort - AOV-Network)

##### DFS

|  |
| --- |
| 207. 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:**   1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs). 2. You may assume that there are no duplicate edges in the input prerequisites.   **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](https://class.coursera.org/algo-003/lecture/52) - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort. 3. Topological sort could also be done via [BFS](http://en.wikipedia.org/wiki/Topological_sorting#Algorithms).   C++1  bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites) {  vector<vector<int>> map(numCourses, vector<int>());  vector<int> count(numCourses);  queue<int> bfsQ;  for(auto prere : prerequisites)  {  map[prere.second].push\_back(prere.first); //map prerequisites course to next courses  count[prere.first]++; //required num of prerequist courses  }  for(int i=0; i<numCourses; ++i)  if(count[i]==0) bfsQ.push(i); //push in the courses wiout prerequisites    int num = 0;  while(!bfsQ.empty())  {  int course = bfsQ.front();  bfsQ.pop();  num++;  for(int nextCourse : map[course])  if(--count[nextCourse] == 0) bfsQ.push(nextCourse); //reduce 1 prerequisite courses from nextCourse  }  return num==numCourses;  }  C++2  bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites) {  vector<vector<int>> graph(numCourses);  vector<int> preNum(numCourses, 0);  stack<int> sq; //DFS  //queue<int> sq; //  int count=0;  //preprocessing  for(auto req : prerequisites)  {  graph[req.second].push\_back(req.first); //collect all limited courses by req.second  preNum[req.first]++; // number of courses needed to be finished before req.first  }  //put avaliable courses in the stack  for(int i=0; i<numCourses; ++i)  {  if(preNum[i]==0) sq.push(i); //avaliable to take  }  //start to take course one by one  while(!sq.empty())  {  int course = sq.top();  sq.pop(); //take one course  count++;  for(auto c : graph[course]) //released courses after taking the course  {  if(--preNum[c]==0) sq.push(c); //no prerequist, avaliable to take  }  }  return count==numCourses;  } |
| 210. 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:**   1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs). 2. You may assume that there are no duplicate edges in the input prerequisites.   [click to show more hints.](https://leetcode.com/problems/course-schedule-ii/)  **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](https://class.coursera.org/algo-003/lecture/52) - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort. 3. Topological sort could also be done via [BFS](http://en.wikipedia.org/wiki/Topological_sorting#Algorithms).   C++1  vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites) {  vector<vector<int>> graph(numCourses);  vector<int> countPrereq(numCourses), res;  stack<int> sta;  for(auto prere : prerequisites)  {  graph[prere.second].push\_back(prere.first);  countPrereq[prere.first]++;  }  for(int i=0; i<numCourses; ++i)  if(countPrereq[i]==0) sta.push(i);  while(!sta.empty())  {  int course = sta.top();  sta.pop();  res.push\_back(course);  for(int next : graph[course])  if(--countPrereq[next]==0) sta.push(next);  }  return res.size()==numCourses ? res : vector<int>();  }  C++2  vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites) {  vector<vector<int>> graph(numCourses);  vector<int> preNum(numCourses), res;  //stack<int> sq; //DFS  queue<int> sq; //BFS  //preprocessing  for(auto req : prerequisites)  {  //collect all limited courses by req.second  graph[req.second].push\_back(req.first);  //number of courses needed to be finished before req.first  preNum[req.first]++;  }  //put avaliable courses in the stack  for(int i=0; i<numCourses; ++i)  {  if(preNum[i]==0) sq.push(i); //avaliable to take  }  //start to take course one by one  while(!sq.empty())  {  //take one course  int course = sq.front();  sq.pop();  res.push\_back(course);  //released courses after taking the course  for(auto c : graph[course])  {  //no prerequist, avaliable to take  if(--preNum[c]==0) sq.push(c);  }  }  return res.size()==numCourses ? res : vector<int>();  } |
|  |
|  |

##### BFS

|  |
| --- |
| 207. 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:**   1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs). 2. You may assume that there are no duplicate edges in the input prerequisites.   [click to show more hints.](https://leetcode.com/problems/course-schedule/)  **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](https://class.coursera.org/algo-003/lecture/52) - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort. 3. Topological sort could also be done via [BFS](http://en.wikipedia.org/wiki/Topological_sorting#Algorithms).   C++1  bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites) {  vector<vector<int>> map(numCourses, vector<int>());  vector<int> count(numCourses);  queue<int> bfsQ;  for(auto prere : prerequisites)  {  map[prere.second].push\_back(prere.first); //map prerequisites course to next courses  count[prere.first]++; //required num of prerequist courses  }  for(int i=0; i<numCourses; ++i)  if(count[i]==0) bfsQ.push(i); //push in the courses wiout prerequisites    int num = 0;  while(!bfsQ.empty())  {  int course = bfsQ.front();  bfsQ.pop();  num++;  for(int nextCourse : map[course])  if(--count[nextCourse] == 0) bfsQ.push(nextCourse); //reduce 1 prerequisite courses from nextCourse  }  return num==numCourses;  }  C++2  bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites) {  vector<vector<int>> graph(numCourses);  vector<int> preNum(numCourses, 0);  stack<int> sq; //DFS  //queue<int> sq; //  int count=0;  //preprocessing  for(auto req : prerequisites)  {  graph[req.second].push\_back(req.first); //collect all limited courses by req.second  preNum[req.first]++; // number of courses needed to be finished before req.first  }  //put avaliable courses in the stack  for(int i=0; i<numCourses; ++i)  {  if(preNum[i]==0) sq.push(i); //avaliable to take  }  //start to take course one by one  while(!sq.empty())  {  int course = sq.top();  sq.pop(); //take one course  count++;  for(auto c : graph[course]) //released courses after taking the course  {  if(--preNum[c]==0) sq.push(c); //no prerequist, avaliable to take  }  }  return count==numCourses;  } |
| 210. 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:**   1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs). 2. You may assume that there are no duplicate edges in the input prerequisites.   [click to show more hints.](https://leetcode.com/problems/course-schedule-ii/)  **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](https://class.coursera.org/algo-003/lecture/52) - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort. 3. Topological sort could also be done via [BFS](http://en.wikipedia.org/wiki/Topological_sorting#Algorithms).   C++1  vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites) {  vector<vector<int>> graph(numCourses);  vector<int> countPrereq(numCourses), res;  stack<int> sta;  for(auto prere : prerequisites)  {  graph[prere.second].push\_back(prere.first);  countPrereq[prere.first]++;  }  for(int i=0; i<numCourses; ++i)  if(countPrereq[i]==0) sta.push(i);  while(!sta.empty())  {  int course = sta.top();  sta.pop();  res.push\_back(course);  for(int next : graph[course])  if(--countPrereq[next]==0) sta.push(next);  }  return res.size()==numCourses ? res : vector<int>();  }  C++2  vector<int> findOrder(int numCourses, vector<pair<int, int>>& prerequisites) {  vector<vector<int>> graph(numCourses);  vector<int> preNum(numCourses), res;  //stack<int> sq; //DFS  queue<int> sq; //BFS  //preprocessing  for(auto req : prerequisites)  {  //collect all limited courses by req.second  graph[req.second].push\_back(req.first);  //number of courses needed to be finished before req.first  preNum[req.first]++;  }  //put avaliable courses in the stack  for(int i=0; i<numCourses; ++i)  {  if(preNum[i]==0) sq.push(i); //avaliable to take  }  //start to take course one by one  while(!sq.empty())  {  //take one course  int course = sq.front();  sq.pop();  res.push\_back(course);  //released courses after taking the course  for(auto c : graph[course])  {  //no prerequist, avaliable to take  if(--preNum[c]==0) sq.push(c);  }  }  return res.size()==numCourses ? res : vector<int>();  } |
| 127. 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 transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.   For example,  Given: *beginWord* = "hit" *endWord* = "cog" *wordList* = ["hot","dot","dog","lot","log","cog"]  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. * You may assume no duplicates in the word list. * You may assume *beginWord* and *endWord* are non-empty and are not the same.   **UPDATE (2017/1/20):** The *wordList* parameter had been changed to a list of strings (instead of a set of strings). Please reload the code definition to get the latest changes.  C++1  int ladderLength(string beginWord, string endWord, vector<string>& wordList) {  //if(wordList.find(endWord) == wordList.end()) return 0;  unordered\_map<string, int> dp;  for(string word : wordList)  dp[word] = INT\_MAX;  if(dp.count(endWord)==0) return 0;  if(beginWord==endWord) return 1;  dp[endWord] = 1;  dp[beginWord] = INT\_MAX;  queue<string> q;  q.push(endWord);  while(!q.empty())  {  string target = q.front();  q.pop();  for(int i=0; i<target.length(); ++i)  {  string temp = target;  for(char c='a'; c<='z'; ++c)  {  if(c == target[i]) continue;  temp[i] = c;  if(dp.count(temp)>0 && dp[temp]==INT\_MAX)  {  dp[temp] = dp[target]+1;  q.push(temp);  }  if(temp==beginWord) return dp[temp];  temp[i] = target[i];  }  }  }  return 0;  }  C++2  int ladderLength(string beginWord, string endWord, unordered\_set<string>& wordList) {  int res=1;  //n = wordList.empty() ? 0 : (\*wordList.begin()).size();  unordered\_set<string> beginSet {beginWord}, endSet{endWord};  while(!beginSet.empty())  {  res++;  unordered\_set<string> set;  for(string word : beginSet) wordList.erase(word);  for(string word : beginSet)  {  string next = word;  for(int i=0; i<word.size(); ++i)  {  for(char c='a'; c<='z'; ++c)  {  next[i] = c;  if(wordList.find(next)==wordList.end()) continue;  if(endSet.find(next)!=endSet.end()) return res;  set.insert(next);  }  next[i] = word[i];  }  }  beginSet = set.size()<endSet.size() ? set : endSet;  endSet = set.size()<endSet.size() ? endSet : set;  }  return 0;  } |
|  |

#### 关键路径(Critical Path - AOE-Network)

#### 回路问题: 欧拉路(Euler Path), 汉密尔顿回路(Hamilton Tour)

#### 差分约束(Difference Constraints): Bellman-Ford

#### 二部图匹配(Bipartite Matching)

#### 网络流(Network Flow)

### 组合数学(Combinatorics)

|  |
| --- |
| Hr nCr table Jim is doing his discrete maths homework which requires him to repeatedly calculate nCr(n choose r) for different values of n. Knowing that this is time consuming, he goes to his sister June for help. June, being a computer science student knows how to convert this into a computer program and generate the answers quickly. She tells him, by storing the lower values of nCr(n choose r), one can calculate the higher values using a very simple formula.  If you are June, how will you calculate nCr values for different values of n?  Since nCr values will be large you have to calculate them modulo 109.  **Input Format**  The first line contains the number of test cases T.  T lines follow each containing an integer n.  **Output Format**  For each n output the list of nC0 to nCn each separated by a single space in a new line. If the number is large, print only the last 9 digits. i.e. modulo 109  **Constraints**  1<=T<=200  1<=n< 1000  **Sample Input**  3  2  4  5  **Sample Output**  1 2 1  1 4 6 4 1  1 5 10 10 5 1  **Explanation**  For 2 we can check 2C0 2C1 and 2C2 are 1, 2 and 1 respectively. The other inputs' answer follow similar pattern.  Java1  public static void main(String[] args) {  int[][] dp = new int[1001][1001];  Initialize(dp);  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  for(int i=0; i<T; ++i){  int n = sc.nextInt();  for(int j=0; j<=n; ++j){  System.out.print(dp[n][j] + " ");  }  System.out.println();  }  }    private static void Initialize(int[][] dp){  for(int i=1; i<=1000; ++i){  for(int j=0; j<=1000; ++j){  if(j==0 || i==j) dp[i][j]=1;  else if(i>j)  dp[i][j] = (dp[i-1][j-1] + dp[i-1][j]) % 1000000000;  }  }  } |
| Hr K Candy Store Jim enters a candy shop which has N different types of candies, each candy is of the same price. Jim has enough money to buy K candies. In how many different ways can he purchase K candies if there are infinite candies of each kind?  **Input Format**  The first line contains an integer T, the number of tests.  This is followed by 2T lines which contain T tests:  The first line (of each testcase) is an integer N and the second line (of each testcase) is an integer K.  **Output Format**  For each testcase, print the number of ways Jim can buy candies from the shop in a newline. If the answer has more than 9 digits, print the last 9 digits.  **Note**  This problem may expect you to have solved [nCr Table](https://www.hackerrank.com/challenges/ncr-table)  **Constraints**  1 <= T <= 200  1 <= N < 1000  1 <= K < 1000  **Sample Input**  2  4  1  2  3  **Sample Output**  4  4  **Explanation**  There are 2 testcases, for the first testcase we have N = 4 and K = 1, as Jim can buy only 1 candy, he can choose to buy any of the 4 types of candies available. Hence, his answer is 4. For the 2nd testcase, we have N = 2 and K = 3, If we name two chocolates as a and b, he can buy  aaa bbb aab abb  chocolates, hence 4.  Java1  public static void main(String[] args) {  int[][] dp = new int[2001][1001];  Initialize(dp);  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  for(int i=0; i<T; ++i){  int N = sc.nextInt();  int K = sc.nextInt();  System.out.println(dp[N+K-1][N-1]);  }  }    private static void Initialize(int[][] dp){  for(int i=1; i<=2000; ++i){  for(int j=0; j<=1000; ++j){  if(j==0 || i==j) dp[i][j]=1;  else if(i>j)  dp[i][j] = (dp[i-1][j-1] + dp[i-1][j]) % 1000000000;  }  }  } |
| HackerRank Diwali Lights  On the eve of [Diwali](https://www.hackerrank.com/external_redirect?to=https://en.wikipedia.org/wiki/Diwali), Hari is decorating his house with a serial light bulb set. The serial light bulb set has N bulbs placed sequentially on a string which is programmed to change patterns every second. If at least one bulb in the set is on at any given instant of time, how many different patterns of light can the serial light bulb set produce?  Note: Lighting two bulbs \*-\* is different from \*\*-  **Input Format**  The first line contains the number of test cases T, T lines follow.  Each line contains an integer N, the number of bulbs in the serial light bulb set.  **Output Format**  Print the total number of patterns modulo 105  **Constraints**  1 <= T <= 1000  0< N < 104  **Sample Input**  2  1  2  **Sample Output**  1  3  **Explanation**  Case 1: 1 bulb can be lit in only 1 way.  Case 2: 2 bulbs can be lit in -\*, \*-, \*\* i.e. 3 ways.  C++1  int CountFormats(int N) {  int res = 1;  for(int i=0; i<N; ++i)  res = (res<<1) % 100000;  return res-1;  }  int main() {  /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/  int T, N;  cin>>T;  for(int i=0; i<T; ++i){  cin>>N;  cout<< CountFormats(N)<<endl;  }  return 0;  }  C++2  int CountFormats(int N) {  if(N==0) return 1;  long res = CountFormats(N/2);  res = (res\*res) % 100000;  return N%2==0 ? res : (res\*2)% 100000;  }  int main() {  /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/  int T, N;  cin>>T;  for(int i=0; i<T; ++i){  cin>>N;  cout<< CountFormats(N) - 1<<endl;  }  return 0;  }  Java1  public static void main(String[] args) {  /\* Enter your code here. Read input from STDIN. Print output to STDOUT. Your class should be named Solution. \*/  Scanner sc = new Scanner(System.in);  int T, N;  T = sc.nextInt();  for(int i=0; i<T && sc.hasNextInt(); ++i){  N = sc.nextInt();  System.out.println(CountFormats(N));  }  }    private static int CountFormats(int N){  int res = 1;  for(int i=0; i<N; ++i)  res = (res<<1) % 100000;  return res-1;  }  Java2  public static void main(String[] args) {  /\* Enter your code here. Read input from STDIN. Print output to STDOUT. Your class should be named Solution. \*/  Scanner sc = new Scanner(System.in);  int T, N;  T = sc.nextInt();  for(int i=0; i<T && sc.hasNextInt(); ++i){  N = sc.nextInt();  System.out.println(CountFormats(N)-1);  }  }    private static long CountFormats(int N){  if(N==0) return 1;  long res = CountFormats(N/2);  res = (res\*res) % 100000;  return N%2==0 ? res : (res\*2) % 100000;  }  **Explanation**  This is a Combinatorics question. We can treat a pattern as selecting X bulbs from N bulbs and put them at N positions. The number of different patterns of X bulbs is which equals .  There are N different selections (1, 2, 3, …, N) in total (0 bulbs is not considered in this question).  Therefore, the total number of patterns is:  Based on Pascal’s Trangle:  If we set x = y = 1, we can get:  Therefore, the answer of the question is -1.  Since 0<N<104, could be extreme large. So, my first idea is N times of multiplying by 2, each time do (result mod 100000). This solution is O(N) time. So, the question is O(T\*N) time in total.  Then, I was thinking the worst-case time complexity of the function (CountPatterns) could be improved. If we use recursion, each time we just need to calculate half of the target n. We just need to take care the cases of odd n, and (res\*res) overflow. This solution is O(log(N)) time and O(log(N)) space to save the recursive results. So, the worst-case time complexity of the question is O(T\*log(N)) time in total. |
| Hr Sherlock And Permutations  Watson asks Sherlock:  Given a string *S* of *N* 0's and *M* 1's, how many unique permutations of this string start with 1?  Help Sherlock by printing the answer modulo (*109+7*).  **Input Format**  First line contains *T*, the number of test cases.  Each test case consists of *N* and *M* separated by a space.  **Output Format**  For each test case, print the answer modulo (*109+7*).  **Constraints**  1 ≤ T ≤ 200  1 ≤ N,M ≤ 1000  **Sample Input**  2  1 1  2 3  **Sample Output**  1  6  **Explanation**  Test1: Out of all unique permutations ie. 01 and 10, only second permutation satisfies. Hence, output is 1.  Test2: Out of all unique permutations ie. 00111 01011 01101 01110 10011 10101 10110 11001 11010 11100, only 10011 10101 10110 11001 11010 11100 satisfy. Hence, output is 6.  C++1  int CountPermutations(int N, int x){  int C[N+1][x+1] = {0};  for(int i=1; i<=N; ++i){  for(int j=0; j<=x; ++j){  if(j==0 || j==i) C[i][j] = 1;  else if(i>j) C[i][j] = (C[i-1][j-1] + C[i-1][j]) % 1000000007;  }  }  return C[N][x];  }  int main() {  int T, N, M;  cin>>T;  for(int i=0; i<T; ++i){  cin>>N>>M;  cout<< CountPermutations(N+M-1, min(N, M-1))<< endl;  }  return 0;  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  int N, M;  for(int i=0; i<T; ++i){  N = sc.nextInt();  M = sc.nextInt();;  System.out.println(combination(N+M-1, Math.min(N, M-1)));  }  }    private static final int MOD = 1000000007;  private static int combination(int N, int x){  int[][] dp = new int[N+1][x+1];  for(int i=1; i<=N; ++i){  for(int j=0; j<=x; ++j){  if(j==0 || i==j) dp[i][j] = 1;  else if(i>j) dp[i][j] = (dp[i-1][j-1] + dp[i-1][j]) % MOD;  }  }  return dp[N][x];  } |
|  |

## 数论(Number Theory)

|  |
| --- |
| 1014 Niven Number |
|  |
|  |
| 367. 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  C++1 (3ms)  bool isPerfectSquare(int num) {  long square = num;  while(square\*square > num)  {  square = (square + num/square) / 2;  }  return square\*square == num;  }  C++2 (2ms)  bool isPerfectSquare(int num) {  if(num==0) return false;  int odd = 1;  while(num>0)  {  num -= odd;  odd += 2;  }  return num==0;  }  Java1 (0ms)  public boolean isPerfectSquare(int num) {  int sqrt = mySqrt(num);  return sqrt\*sqrt == num;  }  private int mySqrt(int num)  {  if(num<=0) return 0;  double err = 1.0e-15;  double temp = num;  while((temp-num/temp) > err\*temp)  {  temp = (temp + num/temp) / 2.0;  }  return (int)temp;  } |
| 69. Sqrt(x)  Implement int sqrt(int x).  Compute and return the square root of *x*.  C++1 (6ms) O(?)  int mySqrt(int x) {  if(x<0) return INT\_MIN;  long n = x;  while(n\*n > x)  n = (n + x/n)/2;  return n;  }  C++2 (8ms) O(log(n))  int mySqrt(int x) {  int left =1, right =x, mid;  while(left<=right)  {  mid = left + (right-left)/2;  if(mid>(x/mid))  {  right = mid-1;  }  else  {  if((mid+1)>x/(mid+1))  {  return mid;  }  left = mid+1;  }  }  return 0;  } |
|  |
| 319. 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 *i*th round, you toggle every *i* bulb. For the *n*th 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.  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  C++1  int bulbSwitch(int n) {  return sqrt(n);  }  C++2  int bulbSwitch(int n) {  int counts = 0;  for (int i=1; i\*i<=n; ++i) {  ++ counts;  }  return counts;  }  Java1  public int bulbSwitch(int n) {  return (int)Math.sqrt(n);  } |

### 素数 Prime

|  |
| --- |
| 263. 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.  C++1  bool isUgly(int num) {  if(num<=0) return false;  int primes[3] = {2,3,5};  for(int i=0; i<3; ++i)  {  if(num % primes[i] == 0)  {  num /= primes[i--];  }  }  return num==1;  } |
| 313. 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 ≤ 106, 0 < primes[i] < 1000. (4) The nth super ugly number is guaranteed to fit in a 32-bit signed integer.  C++1  int nthSuperUglyNumber(int n, vector<int>& primes) {  int pn = primes.size();  vector<int> uglys(n);  vector<int> uglyId(pn);  vector<int> pVal(pn, 1);  int next = 1;  for(int i=0; i<n; ++i)  {  uglys[i] = next;  next = INT\_MAX;  for(int j=0; j<pn; ++j)  {  if(uglys[i] == pVal[j])  pVal[j] = uglys[uglyId[j]++] \* primes[j]; //multiply uglys from beginning  next = min(next, pVal[j]); //update the min val  }  }  return uglys[n-1];  } |
| 264. 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, and *n* **does not exceed 1690**.  C++1 (TLE)  class Solution {  public:  vector<int> factors = {2,3,5};  bool isUgly(int num)  {  for(int i=0; i<3; ++i)  {  if(num % factors[i] == 0)  {  num /= factors[i];  i--;  }  }  return num==1;  }  int nthUglyNumber(int n) {  static vector<int> uglyNums(1,1);  for(int i=uglyNums.back()+1; uglyNums.size()<n ; ++i)  {  if(isUgly(i))  {  uglyNums.push\_back(i);  }  }  return uglyNums[n-1];  }  };  C++2  int nthUglyNumber(int n) {  vector<int> res(1,1);  int p2=0, p3=0, p5=0;  while(res.size()<n)  {  res.push\_back(min(res[p2]\*2, min(res[p3]\*3, res[p5]\*5)));  if(res.back()==res[p2]\*2) ++p2;  if(res.back()==res[p3]\*3) ++p3;  if(res.back()==res[p5]\*5) ++p5;  }  return res[n-1];  }  C++3  int nthUglyNumber(int n) {  static vector<int> uglyNums(1,1);  static int two(0), three(0), five(0);  while(uglyNums.size()<n)  {  int nextMin = min(uglyNums[two]\*2, min(uglyNums[three]\*3, uglyNums[five]\*5));  uglyNums.push\_back(nextMin);  if(uglyNums[two]\*2 == nextMin) two++;  if(uglyNums[three]\*3 == nextMin) three++;  if(uglyNums[five]\*5 == nextMin) five++;  }  return uglyNums[n-1];  } |
| 204. Count Primes  **Description:**  Count the number of prime numbers less than a non-negative number, ***n***.  **Hint:**   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: 4. 2 × 6 = 12 5. 3 × 4 = 12 6. 4 × 3 = 12 7. 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 √*n* because, if *n* is divisible by some number *p*, then *n* = *p* × *q* and since *p* ≤ *q*, we could derive that *p* ≤ √*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 < n; i++) {  if (isPrime(i)) count++;  }  return count;  }  private boolean isPrime(int num) {  if (num <= 1) return false;  // Loop's ending condition is i \* i <= num instead of i <= sqrt(num)  // to avoid repeatedly calling an expensive function sqrt().  for (int i = 2; i \* i <= num; i++) {  if (num % i == 0) return false;  }  return true;  }   1. The [Sieve of Eratosthenes](http://en.wikipedia.org/wiki/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.   https://leetcode.com/static/images/solutions/Sieve_of_Eratosthenes_animation.gif Sieve of Eratosthenes: algorithm steps for primes below 121. "[Sieve of Eratosthenes Animation](http://commons.wikimedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif)" by [SKopp](http://de.wikipedia.org/wiki/Benutzer:SKopp) is licensed under [CC BY 2.0](http://creativecommons.org/licenses/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 × 2 = 6, 3 × 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?   1. 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 × 2 = 10, 5 × 3 = 15, 5 × 4 = 20, 5 × 5 = 25, ... can be marked off. There is a slight optimization here, we do not need to start from 5 × 2 = 10. Where should we start marking off? 2. In fact, we can mark off multiples of 5 starting at 5 × 5 = 25, because 5 × 2 = 10 was already marked off by multiple of 2, similarly 5 × 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 + 2*p*, ... Now what should be the terminating loop condition? 3. 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*? 4. Yes, the terminating loop condition can be *p* < √*n*, as all non-primes ≥ √*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](http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes#Algorithm_complexity).  public int countPrimes(int n) {  boolean[] isPrime = new boolean[n];  for (int i = 2; i < n; i++) {  isPrime[i] = true;  }  // Loop's ending condition is i \* i < n instead of i < sqrt(n)  // to avoid repeatedly calling an expensive function sqrt().  for (int i = 2; i \* i < n; i++) {  if (!isPrime[i]) continue;  for (int j = i \* i; j < n; j += i) {  isPrime[j] = false;  }  }  int count = 0;  for (int i = 2; i < n; i++) {  if (isPrime[i]) count++;  }  return count;  }  C++1 (32ms)  class Solution {  public:  void Clean(vector<int>& count, int prime)  {  for(int i=prime\*prime; i<count.size(); i+=prime)  count[i]=1;  }  int NextPrime(vector<int>& count, int prime)  {  for(int i=prime+1; i<count.size(); ++i)  if(count[i]==0) return i;  return INT\_MAX;  }  int countPrimes(int n) {  if(n<=2) return 0;  vector<int> count(n, 0);  count[0] = count[1] = 1;  int prime = 2;  int maxPrime = sqrt(n);  while(prime <= maxPrime)  {  Clean(count, prime);  prime = NextPrime(count, prime);  }  int primeNum = 0;  for(int i=2; i<n; ++i)  {  if(count[i]==0)  primeNum++;  }  return primeNum;  }  };  C++2 (16ms)  int countPrimes(int n) {  if(--n < 2) return 0;  int m = (n + 1)/2, count = m, k, u = (sqrt(n) - 1)/2;  int notPrime[m] = {0}; //set as 0 save a lot of memory  for(int i = 1; i <= u;i++)  if(!notPrime[i])  for(k = (i+ 1)\*2\*i; k < m;k += i\*2 + 1)  if (!notPrime[k])  {  notPrime[k] = 1;  count--;  }  return count;  }  C++3 (12ms)  int countPrimes(int n) {  if(n < 3) return 0;  int m = n/2, count = m, k, u = (sqrt(n-1)-1)/2;  bool notPrime[m] = {0}; //set as 0 save a lot of memory    for(int i = 1; i <= u;i++)  if(!notPrime[i])  for(k = (i+1)\*2\*i; k < m;k += i\*2 + 1)  if (!notPrime[k])  {  notPrime[k] = true;  count--;  }  return count;  }  Java1 (30ms)  public class Solution {  public int countPrimes(int n) {  if(n<2) return 0;  int[] primes = new int[n];  primes[0]=primes[1] = 1;  int prime = 2;  int m = (int)Math.sqrt(n);  while(prime <= m)  {  cutOff(primes, prime);  prime = nextPrime(primes, prime);  if(prime >= n)  break;  }  int res=0;  for(int b : primes)  {  if(b==0) res++;  }  return res;  }    private void cutOff(int[] primes, int prime)  {  for(int i=prime\*prime; i<primes.length; i+=prime)  primes[i] = 1;  }    private int nextPrime(int[] primes, int prime)  {  prime++;  while(prime<primes.length && primes[prime]==1)  prime++;  return prime;  }  }  Java2 (16ms)  public int countPrimes(int n) {  if(--n < 2) return 0;  int m = (n + 1)/2, count = m, k, u = (int)(Math.sqrt(n) - 1)/2;  int[] notPrime = new int[m];    for(int i = 1; i <= u;i++)  if(notPrime[i]==0)  for(k = (i+ 1)\*2\*i; k < m;k += i\*2 + 1)  if (notPrime[k]==0)  {  notPrime[k] = 1;  count--;  }  return count;  } |
| Codility [CountSemiprimes](https://codility.com/demo/results/training4PMNNH-ZNZ/)  A *prime* is a positive integer X that has exactly two distinct divisors: 1 and X. The first few prime integers are 2, 3, 5, 7, 11 and 13.  A *semiprime* is a natural number that is the product of two (not necessarily distinct) prime numbers. The first few semiprimes are 4, 6, 9, 10, 14, 15, 21, 22, 25, 26.  You are given two non-empty zero-indexed arrays P and Q, each consisting of M integers. These arrays represent queries about the number of semiprimes within specified ranges.  Query K requires you to find the number of semiprimes within the range (P[K], Q[K]), where 1 ≤ P[K] ≤ Q[K] ≤ N.  For example, consider an integer N = 26 and arrays P, Q such that:  P[0] = 1 Q[0] = 26 P[1] = 4 Q[1] = 10 P[2] = 16 Q[2] = 20  The number of semiprimes within each of these ranges is as follows:   * (1, 26) is 10, * (4, 10) is 4, * (16, 20) is 0.   Write a function:  vector<int> solution(int N, vector<int> &P, vector<int> &Q);  that, given an integer N and two non-empty zero-indexed arrays P and Q consisting of M integers, returns an array consisting of M elements specifying the consecutive answers to all the queries.  For example, given an integer N = 26 and arrays P, Q such that:  P[0] = 1 Q[0] = 26 P[1] = 4 Q[1] = 10 P[2] = 16 Q[2] = 20  the function should return the values [10, 4, 0], as explained above.  Assume that:   * N is an integer within the range [1..50,000]; * M is an integer within the range [1..30,000]; * each element of arrays P, Q is an integer within the range [1..N]; * P[i] ≤ Q[i].   Complexity:   * expected worst-case time complexity is O(N\*log(log(N))+M); * expected worst-case space complexity is O(N+M), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **bool** **isPrime**(**int** n)  {  **int** sq = **sqrt**(n);  **for**(**int** i=2; i<=sq; ++i)  **if**(n % i == 0) **return** **false**;  **return** **true**;  }  **bool** **isSemiPrime**(**int** n)  {  **if**(isPrime(n)) **return** **false**;  **int** sq = **sqrt**(n);  **for**(**int** i = sq; i>1; --i)  {  **if**(n%i == 0)  {  **if**(isPrime(i) && isPrime(n/i)) **return** **true**;  **return** **false**;  }  }  **return** **false**;  }  **vector**<**int**> solution(**int** N, **vector**<**int**> &P, **vector**<**int**> &Q) {  // write your code in C++14 (g++ 6.2.0)  **int** maxP = 0;  **for**(**int** q : Q)  **if**(maxP < q) maxP = q;  **vector**<**int**> lSemiPrimes(maxP+1);  **for**(**int** i= 4; i<=maxP; ++i)  {  **if**(isSemiPrime(i)) lSemiPrimes[i]++;  lSemiPrimes[i] += lSemiPrimes[i-1];  }  **int** m = Q.size();  **vector**<**int**> res(m);  **for**(**int** i=0; i<m; ++i)  res[i] = lSemiPrimes[Q[i]] - lSemiPrimes[P[i]-1];  **return** res;  }  Testcase  extreme\_one  small N = 1  [▶](https://codility.com/demo/results/training4PMNNH-ZNZ/)  extreme\_four  small N = 4  [▶](https://codility.com/demo/results/training4PMNNH-ZNZ/)  small\_functional  small functional  [▶](https://codility.com/demo/results/training4PMNNH-ZNZ/)  small\_random  small random, length = ~40  [▶](https://codility.com/demo/results/training4PMNNH-ZNZ/)  medium\_random  small random, length = ~300  [▶](https://codility.com/demo/results/training4PMNNH-ZNZ/)  large\_small\_slices  large with very small slices, length = ~30,000  [▶](https://codility.com/demo/results/training4PMNNH-ZNZ/)  large\_random1  large random, length = ~30,000  [▶](https://codility.com/demo/results/training4PMNNH-ZNZ/)  large\_random2  large random, length = ~30,000  [▶](https://codility.com/demo/results/training4PMNNH-ZNZ/)  extreme\_large  all max ranges |
|  |

1004. Prime Palindromes

### Bit

|  |
| --- |
| 191. Number of 1 Bits  Write a function that takes an unsigned integer and returns the number of ’1' bits it has (also known as the [Hamming weight](http://en.wikipedia.org/wiki/Hamming_weight)).  For example, the 32-bit integer ’11' has binary representation 00000000000000000000000000001011, so the function should return 3.  C++1  int hammingWeight(uint32\_t n) {  int count=0;  while(n > 0)  {  count++;  n &= (n-1);  }  return count;  }  Java1  // you need to treat n as an unsigned value  public int hammingWeight(int n) {  int count=0;  while(n!=0)  {  count++;  n = n&(n-1);  }  return count;  } |
| 190. 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?  C++1 (3ms) O(1)  uint32\_t reverseBits(uint32\_t n) {  uint32\_t res = 0;  for(int i=0; i<32; ++i)  {  res = (res<<1) + (n&1);  n >>= 1;  }  return res;  }  C++2 (3ms)  uint32\_t reverseBits(uint32\_t n) {  uint32\_t reverse = 0;  for(int i=0; i<32; i++)  {  reverse = (reverse<<1) + n%2;  n /= 2;  }  return reverse;  }  Java1 (3ms)  // you need treat n as an unsigned value  public int reverseBits(int n) {  int res=0;  for(int i=0; i<32; ++i)  {  res = (res<<1) + n%2;  n >>>= 1;  }  return res;  } |
| 461. Hamming Distance  The [Hamming distance](https://en.wikipedia.org/wiki/Hamming_distance) between two integers is the number of positions at which the corresponding bits are different.  Given two integers x and y, calculate the Hamming distance.  **Note:** 0 ≤ x, y < 231.  **Example:**  **Input:** x = 1, y = 4  **Output:** 2  **Explanation:**  1 (0 0 0 1)  4 (0 1 0 0)  ↑ ↑  The above arrows point to positions where the corresponding bits are different.  C++1 (6ms) O(1)  int hammingDistance(int x, int y) {  int res = 0;  while(x>0 || y>0)  {  res += (x & 1) ^ (y & 1);  x>>=1;  y>>=1;  }  return res;  }  Python 1  def hammingDistance(self, x, y):  return bin(x^y).count('1') |
| 477. Total Hamming Distance  The [Hamming distance](https://en.wikipedia.org/wiki/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 (just  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.   C++1 (TLE)  class Solution {  public:  int Hamming(int a, int b)  {  int res = 0;  while(a!=0 || b!=0)  {  res += (a&1) ^ (b&1);  a >>= 1;  b >>= 1;  }  return res;  }  int totalHammingDistance(vector<int>& nums) {  int n = nums.size();  if(n<=1) return 0;  vector<int>dp(n);  for(int i=1; i<n; ++i)  {  for(int j=i-1; j>=0; --j)  dp[i-1] += Hamming(nums[i], nums[j]);  dp[i] = dp[i-1];  }  return dp[n-1];  }  };  C++2 (82ms)  int totalHammingDistance(vector<int>& nums) {  int n = nums.size();  if(n<=1) return 0;  int res = 0;  for(int i=0; i<32; ++i)  {  int count1 = 0;  for(auto num : nums)  count1 += (num>>i) & 1;  res += count1 \* (n - count1);  }  return res;  } |
| 476. 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 complement is 0. So you need to output 0.  C++1 (3ms) O(1)  int findComplement(int num) {  int res=0, bit=0;  while(num > 0)  {  res += ((num&1)^1) << bit++;  num >>= 1;  }  return res;  }  Python1  def findComplement(self, num):  i=0  res = 0  while num > 0:  res += ((num&1)^1)<<i  i=i+1  num >>= 1  return res |
| 338. 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.   **Hint:**   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?   C++1 (85ms)  vector<int> countBits(int num) {  vector<int> res(1,0);  while(num>0)  {  int size= res.size();  for(int i=0; i<size && num-->0; ++i)  {  res.push\_back(res[i]+1);  }  }  return res;  }  C++2 (84ms)  vector<int> countBits(int num) {  vector<int> res(num+1,0);  for(int i=1; i<=num; ++i)  {  res[i] = res[i&(i-1)] + 1;  }  return res;  }  C++3 (88ms)  vector<int> countBits(int num) {  vector<int> res(num+1, 0);  if(num==0) return res;  res[1] = 1;  if(num==1) return res;  int div = 1;  for(int i=2; i<=num; ++i)  {  res[i] = res[i%div] + 1;  if(i%div == 0)  div \*= 2;  }  return res;  }  Java1 (2ms)  public int[] countBits(int num) {  int[] res = new int[num+1];  for(int i=1; i<=num; ++i)  {  res[i] = res[i&(i-1)] + 1;  }  return res;  }  Java2 (3ms)  public int[] countBits(int num) {  int[] res = new int[num+1];  if(num==0) return res;  res[1] = 1;  if(num==0) return res;  int div = 1;  for(int i=2; i<=num; ++i)  {  res[i] = res[i%div] + 1;  if(i%div == 0)  div \*= 2;  }  return res;  } |
| 136. 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?  C++1 (16ms)  int singleNumber(vector<int>& nums) {  for(int i=1; i<nums.size(); ++i)  nums[0] ^= nums[i];  return nums[0];  }  def singleNumber(self, nums):  for i in range(1,len(nums)):  nums[0] ^= nums[i]  return nums[0]  Bash1  #!/bin/bash  read N  array=($(cat))  res=0  for i in ${array[@]}  do  res=$(( $res ^ $i ))  done  echo $res  Bash2  #!/bin/bash  read N  array=($(cat))  array=${array[@]}  echo $((${array// /^})) |
| 137. Single Number II  Given an array of integers, every element appears *three* times except for one, which appears exactly once. Find that single one.  **Note:** Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?  The usual bit manipulation code is bit hard to get and replicate. I like to think about the number in 32 bits and just count how many 1s are there in each bit, and sum %= 3 will clear it once it reaches 3. After running for all the numbers for each bit, if we have a 1, then that 1 belongs to the single number, we can simply move it back to its spot by doing res |= sum << i;  This has complexity of O(32n), which is essentially O(n) and very easy to think and implement. Plus, you get a general solution for any times of occurrence. Say all the numbers have 5 times, just do sum %= 5.  C++1 (12ms) O(32n)  int singleNumber(vector<int>& nums) {  int res = 0;  for(int i=0; i<=31; ++i)  {  int sum = 0;  for(auto num : nums)  if((num & (1<<i)) != 0)  sum++;  sum %= 3;  res |= (sum << i);  }  return res;  }  C++2 (12ms)  int singleNumber(vector<int>& nums) {  int ones=0, twos =0;  for(int n : nums)  {  ones = (ones ^ n) & ~twos;  twos = (twos ^ n) & ~ones;  }  return ones;  }  Java1 (6ms)  public int singleNumber(int[] nums) {  int res=0;  for(int i=0; i<32; ++i)  {  int count = 0;  int bit = 1<<i;  for(int num : nums)  if((bit & num) != 0)  count++;  if(count%3 != 0)  res += bit;  }  return res;  }  Testcase:   |  |  |  | | --- | --- | --- | | Id | Testcase | Result | | 1 | [1,1,2,3,3,4,1,2,2,3] | 4 | | 2 | [-1,-1,2,3,3,-4,-1,2,2,3] | -4 | |
| 260. 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?   C++1 (16ms) O(n)  vector<int> singleNumber(vector<int>& nums) {  int key = 0;  vector<int> res(2, 0);  for(int num : nums)  key ^= num;  key = key & (~(key-1));  for(int num : nums)  {  if((num & key) != 0)  res[0] ^= num;  else res[1] ^= num;  }  return res;  }  Testcase:   |  |  |  | | --- | --- | --- | | Id | Testcase | Result | | 1 | [1,1,2,3,3,4] | [2,4] | | 2 | [-1,-1,2,3,3,-4] | [2,-4] | |
| 421. Maximum XOR of Two Numbers in an Array  Given a **non-empty** array of numbers, a0, a1, a2, … , an-1, where 0 ≤ ai < 231.  Find the maximum result of ai XOR aj, where 0 ≤ *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.  C++1 (TLE) O(n2)  int findMaximumXOR(vector<int>& nums) {  int maxV = 0;  for(int i=0; i<nums.size(); ++i)  for(int j=i+1; j<nums.size(); ++j)  maxV = max(maxV, nums[i] xor nums[j]);  return maxV;  }  C++2 O(n)  int findMaximumXOR(vector<int>& nums) {  int max=0, mask=0;  for(int i=31; i>=0; --i)  {  mask |= 1<<i;  unordered\_set<int> set;  for(int num : nums)  set.insert(num & mask);    int temp = max | 1<<i;  for(int profix : set)  {  if(set.count(profix ^ temp)>0)  {  max = temp;  break;  }  }  }  return max;  } |
| 89. 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.  C++1 (4ms) O(n2)  vector<int> grayCode(int n) {  vector<int> res(1);  for(int i=0; i<n; ++i)  for(int j=res.size()-1; j>=0; --j)  res.push\_back(res[j] | (1<<i));  return res;  }  Java1 (3ms)  public List<Integer> grayCode(int n) {  List<Integer> res = new ArrayList<Integer>();  res.add(0);  for(int i=0; i<n; ++i)  for(int j=res.size()-1; j>=0; --j)  res.add((res.get(j) | (1<<i)));  return res;  } |
| 461. Hamming Distance  The [Hamming distance](https://en.wikipedia.org/wiki/Hamming_distance) between two integers is the number of positions at which the corresponding bits are different.  Given two integers x and y, calculate the Hamming distance.  **Note:** 0 ≤ x, y < 231.  **Example:**  **Input:** x = 1, y = 4  **Output:** 2  **Explanation:**  1 (0 0 0 1)  4 (0 1 0 0)  ↑ ↑  The above arrows point to positions where the corresponding bits are different.  C++1  int hammingDistance(int x, int y) {  int res = 0;  while(x>0 || y>0)  {  res += (x & 1) ^ (y & 1);  x>>=1;  y>>=1;  }  return res;  } |
| 476. 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 complement is 0. So you need to output 0.  C++1  int findComplement(int num) {  int res=0, bit=0;  while(num > 0)  {  res += ((num&1)^1) << bit++;  num >>= 1;  }  return res;  } |
| 371. 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.  C++1  int getSum(int a, int b) {  if(b==0) return a;  int carry = (a&b)<<1;  int sum = a^b;  return getSum(sum, carry);  }  C++2  int getSum(int a, int b) {  if(b==0) return a;  return getSum(a^b, (a&b)<<1);  }  Java1  int getSum(int a, int b) {  if(b==0) return a;  return getSum(a^b, (a&b)<<1);  }  Python1  def getSum(self, a, b):  MAX = 0x7fffffff  MIN = 0x80000000  mask = 0xffffffff  while b != 0:  a, b = (a^b) & mask, ((a&b)<<1) & mask  return a if a<=MAX else ~(a ^ mask)  Python2  def getSum(self, a, b):  MAX = 0x7fffffff  MIN = 0x80000000  mask = 0xffffffff  for \_ in range(32):  a, b = a^b, (a&b)<<1  return a if a&MIN > 0 else a & MAX |
| 401. 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.  https://upload.wikimedia.org/wikipedia/commons/8/8b/Binary_clock_samui_moon.jpg  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".   C++1 (0ms)  class Solution {  public:  void TrackingRepresentTimes(vector<string>&res, int LEDs[], int num, int index, int hour, int minute)  {  if(hour>11 || minute>59) return;  if(num == 0)  {  string aTime = to\_string(hour) + ":" + ((minute < 10)? "0" : "") + to\_string(minute);  res.push\_back(aTime);  return;  }  for(int i=index; i<10; ++i)  {  if(i<4)  TrackingRepresentTimes(res, LEDs, num-1, i+1, hour+LEDs[i], minute);  else  TrackingRepresentTimes(res, LEDs, num-1, i+1, hour, minute+LEDs[i]);  }  }  vector<string> readBinaryWatch(int num) {  vector<string> res;  int LEDs[] = {1,2,4,8, 1,2,4,8,16,32};  TrackingRepresentTimes(res, LEDs, num, 0, 0,0);  return res;  }  };  Java1 (3ms)  public class Solution {  public List<String> readBinaryWatch(int num) {  int[] LEDs = {1,2,4,8, 1,2,4,8,16,32};  List<String> res = new ArrayList<String>();  traceTime(res, LEDs, num, 0, 0, 0);  return res;  }  private void traceTime(List<String> res, int[] LEDs, int num, int start, int hour, int minutes)  {  if(hour>11 || minutes>59) return;  if(num == 0)  {  res.add(""+hour+":" + ((minutes<10) ? "0" : "" )+ minutes);  }  else  {  for(int i=start; i<10; ++i)  {  if(i<4) traceTime(res, LEDs, num-1, i+1, hour + LEDs[i], minutes);  else traceTime(res, LEDs, num-1, i+1, hour, minutes + LEDs[i]);  }  }  }  } |
| 405. Convert a Number to Hexadecimal  Given an integer, write an algorithm to convert it to hexadecimal. For negative integer, [two’s complement](https://en.wikipedia.org/wiki/Two%27s_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 0s. 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"  C++1 (0ms)  string toHex(int num) {  if(num==0) return "0";  string res;  for(int count=0; num!=0 && count<8; ++count)  {  int temp = num & 15;  if(temp < 10) res += to\_string(temp);  else res +=('a'+(temp-10));  num = num>>4;  }  reverse(res.begin(), res.end());  return res;  } |
| Codility. Binary Gap  A *binary gap* within a positive integer N is any maximal sequence of consecutive zeros that is surrounded by ones at both ends in the binary representation of N.  For example, number 9 has binary representation 1001 and contains a binary gap of length 2. The number 529 has binary representation 1000010001 and contains two binary gaps: one of length 4 and one of length 3. The number 20 has binary representation 10100 and contains one binary gap of length 1. The number 15 has binary representation 1111 and has no binary gaps.  Write a function:  int solution(int N);  that, given a positive integer N, returns the length of its longest binary gap. The function should return 0 if N doesn't contain a binary gap.  For example, given N = 1041 the function should return 5, because N has binary representation 10000010001 and so its longest binary gap is of length 5.  Assume that:   * N is an integer within the range [1..2,147,483,647].   Complexity:   * expected worst-case time complexity is O(log(N)); * expected worst-case space complexity is O(1).   C++1 (100%)  #**include** <algorithm>  // you can write to stdout for debugging purposes, e.g.  // cout << "this is a debug message" << endl;  **int** **solution**(**int** N) {  // write your code in C++14 (g++ 6.2.0)  **int** oneP = -1, res=0;  **for**(**int** i=0; i<32; ++i)  **if**(((1<<i)&N) >0)  {  **if**(oneP>=0) res = max(res, i-oneP-1);  oneP = i;  }  **return** res;  } |
| 464. 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 >= desiredTotal.  Same with other integers chosen by the first player, the second player will always win.  C++1 (126ms)  class Solution {  public:  unordered\_map<int, bool> map;  bool ifWin(int used, int mci, int dt)  {  if(dt <= 0) return false;  if(map.count(used)==0)  {  for(int i=0; i<mci; ++i)  {  if((1&(used>>i)) == 1) // ((1<<i) & used) > 0  {  if(!ifWin((1<<i)^used, mci, dt-i-1))  {  map[used] = true;  return true;  }  }  }  map[used] = false;  }  return map[used];  }  bool canIWin(int maxChoosableInteger, int desiredTotal) {  int sum = maxChoosableInteger \* (maxChoosableInteger+1) /2;  if(sum < desiredTotal) return false;  if(desiredTotal <= 0) return true;  int used = (1<<maxChoosableInteger) - 1;  return ifWin(used, maxChoosableInteger, desiredTotal);  }  }; |
| 289. Game of Life  According to the [Wikipedia's article](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life): "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](https://en.wikipedia.org/wiki/Moore_neighborhood) (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?   C++1  void gameOfLife(vector<vector<int>>& board) {  int h = board.size();  int w = (h>0) ? board[0].size() : 0;  for(int i=0; i<h; ++i)  {  for(int j=0; j<w; ++j)  {  int lives = 0;  for(int x=((i<1)? 0 :i-1); x<=((i+1)<h ? i+1 : h-1); ++x)  for(int y=((j<1)? 0 :j-1); y<=((j+1)<w ? j+1 : w-1); ++y)  lives += board[x][y] & 1;  if(lives==3 || (lives==4 && board[i][j]==1))  board[i][j] |= 2;  }  }  for(int i=0; i<h; ++i)  for(int j=0; j<w; ++j)  board[i][j] >>= 1;  }  C++2  class Solution {  public:  int lifeInNineCells(vector<vector<int>>& board, int m, int n, int row, int col)  {  int count=0;  for(int i=max(row-1, 0); i<min(row+2,m); ++i)  {  for(int j=max(col-1, 0); j<min(col+2, n); ++j)  count += board[i][j]&1;  }  return count;  }  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 lives = lifeInNineCells(board, m, n, i, j);  if(lives==3 || lives-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;  }  }  }  }; |
| 393. 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.  C++1  class Solution {  public:  int getTypeOfUTF8(int first)  {  if(first<0 || first>=256) return -1;  if((first & 248) == 240) return 3;  if((first & 240) == 224) return 2;  if((first & 224) == 192) return 1;  if((first & 128) == 0) return 0;  return -1;  }  bool isValidedFollower(int follow)  {  if(follow<0 || follow>=256) return false;  return (follow&192) == 128;  }  bool validUtf8(vector<int>& data) {  int i=0, follow=0, n=data.size();  while(i<n)  {  follow = getTypeOfUTF8(data[i]);  if(follow==-1) return false;  while(follow>0)  {  if(i+1==n || !isValidedFollower(data[i+1])) return false;  i++;  follow--;  }  i++;  }  return i==n && follow==0;  }  };  C++2  bool validUtf8(vector<int>& data) {  int count=0;  for(int byte : data)  {  if(count==0)  {  if(byte>>5 == 0b110) count = 1; //C++14 binary  else if(byte>>4 == 0b1110) count = 2;  else if(byte>>3 == 0b11110) count = 3;  else if(byte>>7 == 0b1) return false;  }  else  {  if(byte>>6 != 0b10) return false;  count--;  }  }  return count == 0;  } |
| 201. Bitwise AND of Numbers Range  Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.  For example, given the range [5, 7], you should return 4.  C++1  int rangeBitwiseAnd(int m, int n) {  while(m<n) n = n&(n-1);  return n;  }  C++2 (25ms) beat 95%  int rangeBitwiseAnd(int m, int n) {  if(m>=n) return n;  return rangeBitwiseAnd(m, n&(n-1));  }  C++3  int rangeBitwiseAnd(int m, int n) {  return (m>=n) ? n : rangeBitwiseAnd(m, n&(n-1));  }  C++4  int rangeBitwiseAnd(int m, int n) {  int trans = 0;  while(m!=n)  {  trans++;  m>>=1;  n>>=1;  }  return m<<trans;  }  C++5  int rangeBitwiseAnd(int m, int n) {  return (m < n)? (rangeBitwiseAnd(m/2, n/2)<<1) : m;  } |
| 307. Range Sum Query – Mutable  Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *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.   C++1 (TLE)  class NumArray {  private:  vector<int> nums;  vector<int> sums;  public:  NumArray(vector<int> nums) {  this->nums = nums;  sums = vector<int>(nums.size()+1);  for(int i=nums.size()-1; i>=0; --i)  sums[i] = nums[i] + sums[i+1];  }    void update(int i, int val) {  int change = val - nums[i];  nums[i] = val;  for(int k=i; k>=0; --k)  sums[k] += change;  }    int sumRange(int i, int j) {  return sums[i] - sums[j+1];  }  };  C++2  class NumArray {  private:  vector<int> BIT;  vector<int> Nums;  public:  NumArray(vector<int> &nums){  int n = nums.size();  Nums = nums;  //BIT = vector<int>(n+1);  BIT.resize(n+1);  for(int i=0; i<n; ++i)  {  updateBIT(i, nums[i]);  }  }  void update(int i, int val) {  if(val != Nums[i])  {  updateBIT(i, val-Nums[i]);  Nums[i] = val;  }  }  int sumRange(int i, int j) {  return leftSum(j) - leftSum(i-1);  }    int leftSum(int i)  {  i++;  int sum=0;  while(i>0)  {  sum += BIT[i];  i -= i & (-i);  }  return sum;  }    void updateBIT(int i, int val)  {  i++;  while(i<BIT.size())  {  BIT[i] +=val;  i += i & (-i); //double the last bit  }  }  }; |
| 29. Divide Two Integers  Divide two integers without using multiplication, division and mod operator.  If it is overflow, return MAX\_INT.  C++1  int divide(int dividend, int divisor) {  if(divisor==0 || (dividend==INT\_MIN && divisor==-1)) return INT\_MAX;  int res = 0;  int sign = dividend<0 ^ divisor<0 ? -1 : 1;  long ldend = labs(dividend), lsor = labs(divisor);  while(ldend >= lsor)  {  long temp = lsor, times=1;  while(ldend >= (temp<<1))  {  temp <<= 1;  times <<= 1;  }  ldend -= temp;  res += times;  }  return sign \* res;  } |
| Hr Special Multiple You are given an integer *N*. Can you find the least positive integer *X* made up of only 9's and 0's, such that, X is a multiple of *N*?  **Update**  *X* is made up of one or more occurences of 9 and zero or more occurences of 0.  **Input Format**  The first line contains an integer T which denotes the number of test cases. T lines follow.  Each line contains the integer *N* for which the solution has to be found.  **Output Format**  Print the answer *X* to STDOUT corresponding to each test case. The output should not contain any leading zeroes.  **Constraints**  1 <= T <= 104  1 <= N <= 500  **Sample Input**  3  5  7  1  **Sample Output**  90  9009  9  **Explanation**  90 is the smallest number made up of 9's and 0's divisible by 5. Similarly, you can derive for other cases.  C++1  static vector<long> mem;  long countSmallest90Num(int n){  for(int i=0; i<mem.size(); ++i){  if(mem[i] % n == 0)  return mem[i];  }  int next = mem.size();  while(++next){  string bina;  int x = next;  while(x>0){  bina = to\_string(x&1) + bina;  x >>= 1;  }  replace(bina.begin(), bina.end(), '1', '9');  long res = stol(bina, nullptr, 10);  mem.push\_back(res);  if(res % n == 0)  return res;  }  return 0;  }  int main() {  int T, N;  cin>>T;  for(int i=0; i<T; ++i){  cin>>N;  cout<<countSmallest90Num(N)<<endl;  }  return 0;  }  Java1  private static List<Long> mem = new LinkedList<Long>();    public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  for(int i=0; i<T; ++i){  int N = sc.nextInt();  System.out.println(FindSmallest90Num(N));  }  }    private static long FindSmallest90Num(int N){  for(int i=0; i<Solution.mem.size(); ++i){  if(Solution.mem.get(i) % N == 0)  return Solution.mem.get(i);  }  int next = Solution.mem.size();  while(++next > 0){  String binaryI = Integer.toBinaryString(next);  binaryI = binaryI.replace('1', '9');  long res = Long.parseLong(binaryI);  Solution.mem.add(res);  if(res % N == 0) return res;  }  return 0;  } |

### 最大公约数Greatest Common Divisor (GCD)

|  |
| --- |
| Codility [ChocolatesByNumbers](https://codility.com/demo/results/training7355RW-YX6/)  Two positive integers N and M are given. Integer N represents the number of chocolates arranged in a circle, numbered from 0 to N − 1.  You start to eat the chocolates. After eating a chocolate you leave only a wrapper.  You begin with eating chocolate number 0. Then you omit the next M − 1 chocolates or wrappers on the circle, and eat the following one.  More precisely, if you ate chocolate number X, then you will next eat the chocolate with number (X + M) modulo N (remainder of division).  You stop eating when you encounter an empty wrapper.  For example, given integers N = 10 and M = 4. You will eat the following chocolates: 0, 4, 8, 2, 6.  The goal is to count the number of chocolates that you will eat, following the above rules.  Write a function:  int solution(int N, int M);  that, given two positive integers N and M, returns the number of chocolates that you will eat.  For example, given integers N = 10 and M = 4. the function should return 5, as explained above.  Assume that:   * N and M are integers within the range [1..1,000,000,000].   Complexity:   * expected worst-case time complexity is O(log(N+M)); * expected worst-case space complexity is O(log(N+M)).   C++1 (75%)  #**include** <unordered\_map>  // you can write to stdout for debugging purposes, e.g.  // cout << "this is a debug message" << endl;  std::**unordered\_map**<**int**, **int**> **map**;  **int** **myRecursive**(**int** start, **int** N, **int** M)  {  **if**(**map**[start]>0) **return** 0;  **map**[start] = 1;  **return** myRecursive((start+M)%N, N, M) + 1;  }  **int** **solution**(**int** N, **int** M) {  // write your code in C++14 (g++ 6.2.0)  **if**(M==1) **return** N;  **return** myRecursive(0, N, M);  }  C++2  **int** **GCD**(**int** N, **int** M)  {  **if**(N < M) **return** GCD(M, N);  **if**(M==0) **return** N;  **return** GCD(M, N%M);  }  **int** **solution**(**int** N, **int** M) {  // write your code in C++14 (g++ 6.2.0)  **return** N / GCD(N, M);  }  C++3  **int** **GCD**(**int** N, **int** M)  {  **return** (M==0) ? N : GCD(M, N%M);  }  **int** **solution**(**int** N, **int** M) {  // write your code in C++14 (g++ 6.2.0)  **return** N / GCD(N, M);  }  Java1  **class Solution {**  **public** **int** gcd(**int** a, **int** b) {  **if** ((a % b) == 0) {  **return** b;  } **else** {  **return** gcd(b, a % b);  }  }  **public** **int** solution(**int** N, **int** M) {  **return** N / gcd(N, M);  }  } |
| Codility [CommonPrimeDivisors](https://codility.com/demo/results/trainingEKGBD3-MWX/)  A *prime* is a positive integer X that has exactly two distinct divisors: 1 and X. The first few prime integers are 2, 3, 5, 7, 11 and 13.  A prime D is called a *prime divisor* of a positive integer P if there exists a positive integer K such that D \* K = P. For example, 2 and 5 are prime divisors of 20.  You are given two positive integers N and M. The goal is to check whether the sets of prime divisors of integers N and M are exactly the same.  For example, given:   * N = 15 and M = 75, the prime divisors are the same: {3, 5}; * N = 10 and M = 30, the prime divisors aren't the same: {2, 5} is not equal to {2, 3, 5}; * N = 9 and M = 5, the prime divisors aren't the same: {3} is not equal to {5}.   Write a function:  int solution(vector<int> &A, vector<int> &B);  that, given two non-empty zero-indexed arrays A and B of Z integers, returns the number of positions K for which the prime divisors of A[K] and B[K] are exactly the same.  For example, given:  A[0] = 15 B[0] = 75 A[1] = 10 B[1] = 30 A[2] = 3 B[2] = 5  the function should return 1, because only one pair (15, 75) has the same set of prime divisors.  Assume that:   * Z is an integer within the range [1..6,000]; * each element of arrays A, B is an integer within the range [1..2,147,483,647].   Complexity:   * expected worst-case time complexity is O(Z\*log(max(A)+max(B))2); * expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **GCD**(**int** A, **int** B)  {  **return** B==0 ? A : GCD(B, A%B);  }  **int** **divideByGCD**(**int** A, **int** gcd)  {  **int** c = GCD(A, gcd);  **return** (c==1) ? A : divideByGCD(A/c, gcd);  }  **bool** **hasSamePrimeDivisort**(**int** A, **int** B)  {  **int** gcd = GCD(A, B);  **int** a = divideByGCD(A, gcd);  **int** b = divideByGCD(B, gcd);  **return** a==1 && b==1;  }  **int** **solution**(**vector**<**int**> &A, **vector**<**int**> &B) {  // write your code in C++14 (g++ 6.2.0)  **int** res = 0;  **for**(**unsigned** **int** i=0; i<A.size(); ++i)  **if**(hasSamePrimeDivisort(A[i], B[i]))  res++;  **return** res;  }  Testcase  extreme  extreme test with small values  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  simple\_1  simple test with small values  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  simple\_2  simple test with small values  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  primes  powers of primes  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  small\_primes  small primes  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  small\_all\_pairs  all pairs 1-10, length = 100  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  small\_random  small random test, length = 100  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  large\_all\_pairs  all pairs 1-70, length = ~5,000  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  large\_random  large random tests, length = ~6,000  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  many\_factors  factorial test  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  many\_factors2  factorial test  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  big\_powers  powers of 2 and 3  [▶](https://codility.com/demo/results/trainingEKGBD3-MWX/)  extreme\_maximal  extreme test with maximal values |
| 365. 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](https://www.youtube.com/watch?v=BVtQNK_ZUJg))  Input: x = 3, y = 5, z = 4  Output: True  **Example 2:**  Input: x = 2, y = 6, z = 5  Output: False  C++1  class Solution {  public:  int GCD(int x, int y)  {  if(y==0) return x;  return GCD(y, x%y);  }  bool canMeasureWater(int x, int y, int z) {  if(z==0) return true;  return z<=(long)x+y && z%GCD(x,y)==0;  }  };  C++2  class Solution {  public:  int GCD(int x, int y)  {  return y==0 ? x : GCD(y, x%y);  }  bool canMeasureWater(int x, int y, int z) {  return z==0 || z <=(long long)x + y && z%GCD(x,y)==0;  }  }; |
| Hr Restaurant Martha is interviewing at Subway. One of the rounds of the interview requires her to cut a bread of size l×b into smaller identical pieces such that each piece is a square having maximum possible side length with no left over piece of bread.  **Input Format**  The first line contains an integer T. T lines follow. Each line contains two space separated integers l and b which denote length and breadth of the bread.  **Constraints**   * 1 <= T <= 1000 * 1 <= l,b <= 1000   **Output Format**  T lines, each containing an integer that denotes the number of squares of maximum size, when the bread is cut as per the given condition.  **Sample Input 0**  2  2 2  6 9  **Sample Output 0**  1  6  **Explanation 0**  The 1st testcase has a bread whose original dimensions are 2×2, the bread is uncut and is a square. Hence the answer is 1.  The 2nd testcase has a bread of size 6×9. We can cut it into 54 squares of size 1×1, 6 of size 3×3. For other sizes we will have leftovers. Hence, the number of squares of maximum size that can be cut is 6.  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  int l, b, gcd;  for(int i=0; i<T; ++i){  l = sc.nextInt();  b = sc.nextInt();  gcd = GCD(l, b);  System.out.println((l\*b)/(gcd\*gcd));  }  }  private static int GCD(int a, int b){  return b==0 ? a : GCD(b, a%b);  } |
| Hr Sumar and the Floating Rocks Famous wizard Sumar moonji kumaru is stuck in a huge room and has to save Hermione Granger from a monster. Kumaru is at location P1 given by integral coordinates (x1,y1) and Hermione is at location P2 given by integral coordinates (x2,y2). Sadly P1 and P2 are the only points at which floating rocks are present. Rest of the room is without floor and underneath is hot lava.  Kumaru has to go from P1 to P2 but there are no floating rocks to walk on. Kumaru knows a spell that can make the rocks appear but only on the integral coordinates on the straight line joining P1 and P2.  How many rocks can appear at locations (x,y) on the line segment between P1 and P2 (excluding P1 and P2) which satisfy the condition that both x and y are integers?  **Input Format**  The first line contains a single integer T, the number of test cases. T lines follow.  Each of the following T lines contains one test case each. Each test case contains 4 integers x1, y1, x2 and y2separated by a single space.  **Output Format**  A single line containing the number of rocks.  **Constraints**  1 <= T <= 105  -109 <= x1, y1, x2, y2 <= 109  **Sample input**  3  0 2 4 0  2 2 5 5  1 9 8 16  **Sample Output**  1  2  6  **Explanation**  2Dplane  Case 1: As shown in the figure, between (0,2) and (4,0) there's only 1 integral point (2,1) hence 1 rock.  Case 2: Between (2,2) and (5,5) lies (3,3) and (4,4), hence 2 rocks.  Case 3: Between (1,9) and (8,16) there lies 6 rocks at positions (2,10) (3,11) (4,12) (5,13) (6,14) (7,15).  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  int[] p = new int[4];  for(int i=0; i<T; ++i){  for(int j=0; j<4; ++j)  p[j] = sc.nextInt();  int gcd = GCD(p[0]-p[2], p[1]-p[3]);  System.out.println(Math.abs(gcd)-1);  }  }    private static int GCD(int x, int y){  return y==0 ? x : GCD(y, x%y);  } |
| Hr Die Hard 3 **Simon**: On the fountain, there should be 2 jugs - a 5 gallon and a 3 gallon. Do you see them? Fill one of the jugs with exactly 4 gallons of water and place it on the scale and the timer will stop. You must be precise; one ounce more or less will result in detonation. If you’re still alive in 5 minutes, we’ll speak.  **Bruce**: Wait, wait a second. I don’t get it. Do you get it?  **Samuel**: No.  **Bruce**: Get the jugs. Obviously, we can’t ﬁll the 3 gallon jug with 4 gallons of water.  **Samuel**: Obviously.  **Bruce**: All right. I know, here we go. We ﬁll the 3 gallon jug exactly to the top, right?  **Samuel**: Uh huh.  **Bruce**: Okay, now we pour this 3 gallons into the 5 gallon jug, giving us exactly 3 gallons in the 5 gallon jug, right?  **Samuel**: Right, then what?  **Bruce**: We take the 3 gallon jug and ﬁll it a third of the way...  **Samuel**: No! He said, “Be precise.” Exactly 4 gallons.  **Bruce**: Damn! Every cop within 50 miles is running his ass off and I’m out here playing a kids' games in the park.  **Samuel**: Hey, you want to focus on the problem at hand?  Given 2 jugs of capacity *a* and *b* gallons, and an infinite supply of water, can you fill one of the jugs with exactly *c* gallons of water ?  **Input Format**  First line contains the number of testcases *T*. *T* lines follow.  Each line contains 3 space separated integers *a*, *b* and *c* . *a* and *b* indicate the capacity of the two jugs respectively, and *c* denotes the exact capacity with which one of the jugs should be filled.  **Output Format**  For each test case, print "YES" (in a new line) if one of the jugs can be filled with exactly *c* gallons of water and "NO" (in a new line) if they cannot be filled. ( quotes are for clarity )  **Constraints**  1 ≤ a, b, c ≤ 103  1 ≤ T ≤ 100  **Sample Input**  2  5 3 4  3 6 4  **Sample Output**  YES  NO  **Explanation**  Bruce can do the following, fill jug *a* with 5 gallons.  a = 5, b = 0  Now, he can fill jug *b* with 3 gallons from jug *a*.  a = 2, b = 3  He can empty jug *b* and empty 2 gallons from jug *a* to jug *b*.  a = 0, b = 2  Now, he can fill jug *a* with 5 gallons and fill jug *b* with 1 gallon from jug *a*. This results in jug *a* containing exactly 4 gallons of water.  a = 5, b = 2  a = 4, b = 3  In the second testcase, it is impossible to generate 4 gallons of water, hence NO.  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  int a, b, c;  for(int i=0; i<T; ++i){  a = sc.nextInt();  b = sc.nextInt();  c = sc.nextInt();  System.out.println(CanFill(a, b, c));  }  }    private static String CanFill(int a, int b, int c){  int gcd = GCD(a, b);  if((c>a && c>b) || c%gcd!=0) return "NO";  return "YES";  }    private static int GCD(int a, int b){  return b==0 ? a : GCD(b, a%b);  } |
| Hr Possible Path Adam is standing at point (a,b) in an infinite 2D grid. He wants to know if he can reach point (x,y or not. The only operation he can do is to move to point (a+b,b), (a-b, b), (a, b+a), or (a, b-a) from some point (a, b). It is given that he can move to any point on this 2D grid, i.e., the points having positive or negative X(or Y) co-ordinates.  Tell Adam whether he can reach (x, y) or not.  **Input Format**  The first line contains an integer, T, followed by T lines, each containing 4 space-separated integers i.e. a, b, x and y.  **Constraints**   * 1 <= T <= 1000 * 1 <= a, b, x, y <= 1018   **Output Format**  For each test case, display YES or NO that indicates if Adam can reach (x, y) or not.  **Sample Input**  3  1 1 2 3  2 1 2 3  3 3 1 1  **Sample Output**  YES  YES  NO  **Explanation**   1. (1,1) -> (2,1) -> (2,3).   Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  long a, b, x, y;  for(int i=0; i<T; ++i){  a = sc.nextLong();  b = sc.nextLong();  x = sc.nextLong();  y = sc.nextLong();  System.out.println(CanReach(a, b, x, y));  }  }    private static String CanReach(long a, long b, long x, long y){  if(GCD(a,b) == GCD(x,y))  return "YES";  return "NO";  }    private static long GCD(long a, long b){  return b==0 ? a : GCD(b, a%b);  } |
|  |

### 最小公倍数Least Common Demominator (LCD)

|  |
| --- |
| 592. Fraction Addition and Subtraction  Given a string representing an expression of fraction addition and subtraction, you need to return the calculation result in string format. The final result should be [irreducible fraction](https://en.wikipedia.org/wiki/Irreducible_fraction). If your final result is an integer, say 2, you need to change it to the format of fraction that has denominator 1. So in this case, 2 should be converted to 2/1.  **Example 1:**  **Input:**"-1/2+1/2"  **Output:** "0/1"  **Example 2:**  **Input:**"-1/2+1/2+1/3"  **Output:** "1/3"  **Example 3:**  **Input:**"1/3-1/2"  **Output:** "-1/6"  **Example 4:**  **Input:**"5/3+1/3"  **Output:** "2/1"  **Note:**   1. The input string only contains '0' to '9', '/', '+' and '-'. So does the output. 2. Each fraction (input and output) has format ±numerator/denominator. If the first input fraction or the output is positive, then '+' will be omitted. 3. The input only contains valid **irreducible fractions**, where the **numerator** and **denominator** of each fraction will always be in the range [1,10]. If the denominator is 1, it means this fraction is actually an integer in a fraction format defined above. 4. The number of given fractions will be in the range [1,10]. 5. The numerator and denominator of the **final result** are guaranteed to be valid and in the range of 32-bit int.   C++1  class Solution {  public:  int GCD(int a, int b)  {  return b==0 ? a : GCD(b, a%b);  }  int MCD(int a, int b)  {  int gcd = GCD(a, b);  return a/gcd \* b;  }  void add(int (&pre)[2], int cur[2])  {  int mcd = MCD(pre[1], cur[1]);  pre[0] = mcd/pre[1] \* pre[0] + mcd/cur[1] \* cur[0];  pre[1] = mcd;  int gcd = GCD(abs(pre[0]), pre[1]);  pre[0] /= gcd;  pre[1] /= gcd;  }  string fractionAddition(string expression) {  if(expression.empty() || expression=="+" || expression=="-") return "0/1";  int sign = expression[0]=='-' ? -1 : 1;  int num=0, n = expression.length();  int pre[2] = {0,1}, cur[2] = {0,1};  //vector<int> pre(vals, vals+2), cur(vals, vals+2);  expression += "+";  for(int i= (expression[0]=='-' ? 1 : 0); i<=n; ++i)  {  if(isdigit(expression[i]))  num = 10\*num + (expression[i]-'0');  else if(expression[i] == '/')  {  cur[0] = sign\*num;  num = 0;  }  else  {  cur[1] = num;  num=0;  sign = expression[i]=='-' ? -1 : 1;  add(pre, cur);  }  }  return to\_string(pre[0]) + "/" + to\_string(pre[1]);  }  }; |
|  |
|  |

### 同余

當两个[整数](https://zh.wikipedia.org/wiki/%E6%95%B4%E6%95%B0)[除](https://zh.wikipedia.org/wiki/%E5%B8%A6%E4%BD%99%E9%99%A4%E6%B3%95)以同一个正整数，若得相同[余数](https://zh.wikipedia.org/wiki/%E4%BD%99%E6%95%B0)，则二整数同余

### Factorial

|  |
| --- |
| 172. Factorial Trailing Zeroes  Given an integer *n*, return the number of trailing zeroes in *n*!.  **Note:**Your solution should be in logarithmic time complexity.  C++ (3ms)  int trailingZeroes(int n) {  return (n==0) ? 0 : n/5 + trailingZeroes(n/5);  } |
|  |
|  |

### Fibonacci

|  |
| --- |
|  |
| 70. Climbing Stairs  You are climbing a stair case. It takes *n* steps to reach to the top.  Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?  **Note:** Given *n* will be a positive integer.  C++1 (0ms)  int climbStairs(int n) {  if(n<2) return 1;  int f1=1, f2=1, res=0;  for(int i=2;i<=n;i++)  {  res = f1+f2;  f1=f2;  f2=res;  }  return res;  }  Java1 (TLE)  public int climbStairs(int n) {  if(n<2) return 1;  else if(n==2) return 2;  else return climbStairs(n-1) + climbStairs(n-2);  } |
| Codility [Ladder](https://codility.com/demo/results/training3BUAUM-NNB/)  You have to climb up a ladder. The ladder has exactly N rungs, numbered from 1 to N. With each step, you can ascend by one or two rungs. More precisely:   * with your first step you can stand on rung 1 or 2, * if you are on rung K, you can move to rungs K + 1 or K + 2, * finally you have to stand on rung N.   Your task is to count the number of different ways of climbing to the top of the ladder.  For example, given N = 4, you have five different ways of climbing, ascending by:   * 1, 1, 1 and 1 rung, * 1, 1 and 2 rungs, * 1, 2 and 1 rung, * 2, 1 and 1 rungs, and * 2 and 2 rungs.   Given N = 5, you have eight different ways of climbing, ascending by:   * 1, 1, 1, 1 and 1 rung, * 1, 1, 1 and 2 rungs, * 1, 1, 2 and 1 rung, * 1, 2, 1 and 1 rung, * 1, 2 and 2 rungs, * 2, 1, 1 and 1 rungs, * 2, 1 and 2 rungs, and * 2, 2 and 1 rung.   The number of different ways can be very large, so it is sufficient to return the result modulo 2P, for a given integer P.  Write a function:  vector<int> solution(vector<int> &A, vector<int> &B);  that, given two non-empty zero-indexed arrays A and B of L integers, returns an array consisting of L integers specifying the consecutive answers; position I should contain the number of different ways of climbing the ladder with A[I] rungs modulo 2B[I].  For example, given L = 5 and:  A[0] = 4 B[0] = 3 A[1] = 4 B[1] = 2 A[2] = 5 B[2] = 4 A[3] = 5 B[3] = 3 A[4] = 1 B[4] = 1  the function should return the sequence [5, 1, 8, 0, 1], as explained above.  Assume that:   * L is an integer within the range [1..30,000]; * each element of array A is an integer within the range [1..L]; * each element of array B is an integer within the range [1..30].   Complexity:   * expected worst-case time complexity is O(L); * expected worst-case space complexity is O(L), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **vector**<**int**> getLadders(**int** L, **int** mod)  {  mod = **pow**(2, mod);  **vector**<**int**> ladders(L+1);  ladders[1] = 1;  **if**(L<=1) **return** ladders;  ladders[2] = 2;  **int** a = 1, b = 2;  **for**(**int** i=3; i<=L; ++i)  {  ladders[i] = (a + b) % mod;  a = b;  b = ladders[i];  }  **return** ladders;  }  **vector**<**int**> solution(**vector**<**int**> &A, **vector**<**int**> &B) {  // write your code in C++14 (g++ 6.2.0)  **int** L = 0;  **for**(**int** a : A)  **if**(a > L) L = a;  **vector**<**int**> ladders(L+1);  ladders = getLadders(L, 30);  **int** Q = A.size();  **vector**<**int**> res(Q);  **for**(**int** i=0; i<Q; ++i)  res[i] = ladders[A[i]] % (**int**)(**pow**(2,B[i]));  **return** res;  }  Testcase  extreme  extreme small values  [▶](https://codility.com/demo/results/training3BUAUM-NNB/)  small\_functional  small functional  [▶](https://codility.com/demo/results/training3BUAUM-NNB/)  small  small tests  [▶](https://codility.com/demo/results/training3BUAUM-NNB/)  small\_random  small random, length = ~100  [▶](https://codility.com/demo/results/training3BUAUM-NNB/)  medium\_random  medium random, length = ~1,000  [▶](https://codility.com/demo/results/training3BUAUM-NNB/)  large\_range  large range, length = ~30,000  [▶](https://codility.com/demo/results/training3BUAUM-NNB/)  large\_random  large random, length = ~30,000  [▶](https://codility.com/demo/results/training3BUAUM-NNB/)  extreme\_large  all max size of the ladder |
| Codility [FibFrog](https://codility.com/demo/results/trainingGVWPAS-82V/)  The Fibonacci sequence is defined using the following recursive formula:  F(0) = 0 F(1) = 1 F(M) = F(M - 1) + F(M - 2) if M >= 2  A small frog wants to get to the other side of a river. The frog is initially located at one bank of the river (position −1) and wants to get to the other bank (position N). The frog can jump over any distance F(K), where F(K) is the K-th Fibonacci number. Luckily, there are many leaves on the river, and the frog can jump between the leaves, but only in the direction of the bank at position N.  The leaves on the river are represented in a zero-indexed array A consisting of N integers. Consecutive elements of array A represent consecutive positions from 0 to N − 1 on the river. Array A contains only 0s and/or 1s:   * 0 represents a position without a leaf; * 1 represents a position containing a leaf.   The goal is to count the minimum number of jumps in which the frog can get to the other side of the river (from position −1 to position N). The frog can jump between positions −1 and N (the banks of the river) and every position containing a leaf.  For example, consider array A such that:  A[0] = 0 A[1] = 0 A[2] = 0 A[3] = 1 A[4] = 1 A[5] = 0 A[6] = 1 A[7] = 0 A[8] = 0 A[9] = 0 A[10] = 0  The frog can make three jumps of length F(5) = 5, F(3) = 2 and F(5) = 5.  Write a function:  int solution(vector<int> &A);  that, given a zero-indexed array A consisting of N integers, returns the minimum number of jumps by which the frog can get to the other side of the river. If the frog cannot reach the other side of the river, the function should return −1.  For example, given:  A[0] = 0 A[1] = 0 A[2] = 0 A[3] = 1 A[4] = 1 A[5] = 0 A[6] = 1 A[7] = 0 A[8] = 0 A[9] = 0 A[10] = 0  the function should return 3, as explained above.  Assume that:   * N is an integer within the range [0..100,000]; * each element of array A is an integer that can have one of the following values: 0, 1.   Complexity:   * expected worst-case time complexity is O(N\*log(N)); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **struct** Point{  **int** x;  **int** step;  Point(**int** a, **int** b) : x(a), step(b) {}  };  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size();  **vector**<**int**> fib(1);  fib.push\_back(1);  **for**(**int** i=2; fib.back()<=n; ++i)  fib.push\_back(fib[i-1] + fib[i-2]);    **queue**<Point> q ( {Point(-1, 0)} );  **while**(!q.empty())  {  Point point = q.front();  q.pop();  **for**(**unsigned** **int** i = fib.size()-1; i>=2; --i)  {  **int** next = point.x + fib[i];  **if**(next == n) **return** point.step + 1;  **if**(next > n || next <0 || A[next]==0) **continue**;  q.push(Point(next, point.step+1));  A[next] = 0;  }  }  **return** A.empty() ? 1 : -1;  }  Testcase  extreme\_small\_ones  empty array : get -1 expected 1  and all ones  ✘  WRONG ANSWER  got 2 expected 1  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  extreme\_small\_zeros  all zeros  ✘  WRONG ANSWER  got -1 expected 1  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  simple\_functional  simple functional tests  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  small\_random  small random test, length = ~100  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  small\_cyclic  small cyclic test, length = ~500  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  small\_fibonacci  small Fibonacci word test, length = 610  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  medium\_random  medium random test, length = ~5,000  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  medium\_thue\_morse  medium Thue-Morse sequence, lenght = 2^13  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  large\_big\_result  large test with big result, length = ~100,000  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  large\_cyclic  large cyclic test, length = ~100,000  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  large\_random  large random test, length = ~100,000  [▶](https://codility.com/demo/results/trainingENB85Y-7VZ/)  extreme\_large\_ones\_zeros  all zeros / ones, length = ~100,000 |
| Hr Is Fibo You are given an integer, N. Write a program to determine if N is an element of the *Fibonacci sequence*.  The first few elements of the Fibonacci sequence are 0,1,1,2,3,5,8,13,…. A Fibonacci sequence is one where every element is a sum of the previous two elements in the sequence. The first two elements are 0 and 1.  Formally:  Fib0 = 0  Fib1 = 1  ….  Fibn = Fibn-1 + Fibn-2  (n>1)  **Input Format**  The first line contains N, number of test cases.  N lines follow. Each line contains an integer N.  **Output Format**  Display IsFibo if N is a Fibonacci number and IsNotFibo if it is not. The output for each test case should be displayed in a new line.  **Constraints**   1 <= T <= 105 1 <= N <= 1010  **Sample Input**  3  5  7  8  **Sample Output**  IsFibo  IsNotFibo  IsFibo  **Explanation**  5 is a Fibonacci number given by Fib5 = 3+2  7 is not a Fibonacci number  8 is a Fibonacci number given by Fib8 = 3+5  C++1  static set<long> mem({0, 1, 2});  string isFibo(long N){  if(mem.count(N)>0) return "IsFibo";  else return "IsNotFibo";  }  int main() {  int T;  long N;  cin>>T;  for(int i=mem.size(); i<60; ++i){  long temp = \*(mem.rbegin()) + \*(++mem.rbegin());  mem.insert(temp);  }  for(int i=0; i<T; ++i){  cin>>N;  cout<<isFibo(N)<<endl;  }  return 0;  }  Java1  static LinkedList<Long> mem = new LinkedList<Long>(Arrays.asList((long)1, (long)2));  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  for(int i=0; i<T; ++i){  long N = sc.nextLong();  System.out.println(isFibonacci(N));  }  }    private static String isFibonacci(long N){  if(Solution.mem.contains(N)) return "IsFibo";  for(int i=Solution.mem.size(); Solution.mem.getLast()<N; ++i){  long temp = Solution.mem.getLast() + Solution.mem.get(i-2);  Solution.mem.add(temp);  if(temp==N) return "IsFibo";  }  return "IsNotFibo";  } |
|  |
|  |
|  |

### Power

|  |
| --- |
| 231. Power of Two  Given an integer, write a function to determine if it is a power of two.  C++1 (3ms)  bool isPowerOfTwo(int n) {  if(n<=0) return false;  while(n != 1)  {  if(n%2 != 0) return false;  n /= 2;  }  return true;  }  C++2 (3ms)  bool isPowerOfTwo(int n) {  if(n<=0) return false;  if(n==1) return true;  return (n%2==0) ? isPowerOfTwo(n/2) : false;  }  C++3 (3ms)  bool isPowerOfTwo(int n) {  return fmod(log10(n)/log10(2), 1) < 1e-15;  }  C++4 (3ms)  bool isPowerOfTwo(int n) {  return n>0 && (n&(n-1))==0;  }  Java1 (3ms)  public boolean isPowerOfTwo(int n) {  return Math.log(n)/Math.log(2) % 1 < 1e-9;  } |
| 326. 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?  C++1 (55ms)  bool isPowerOfThree(int n) {  //return fmod(log10(n)/log10(3),1) <1e-9;  return fmod(log10(n)/log10(3),1) == 0;  }  C++2 (95ms)  bool isPowerOfThree(int n) {  if(n<1) return false;  if(n==1) return true;  while(n>1)  {  if(n%3 != 0) return false;  n /= 3;  }  return true;  }  C++3 (49ms)  bool isPowerOfThree(int n) {  if(n<1) return false;  if(n==1) return true;  return (n%3 != 0)? false : isPowerOfThree(n/3);  }  Java1 (18ms)  public boolean isPowerOfThree(int n) {  return n>0 && 1162261467%n==0;  }  Java2 (19ms)  public boolean isPowerOfThree(int n) {  if(n==0) return false;  while(n%3==0) n/=3;  return n==1;  } |
| 342. 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?  C++1 (3ms)  bool isPowerOfFour(int num) {  return fmod(log10(num)/log10(4), 1) < 1e-16;  }  C++2 (8ms)  bool isPowerOfFour(int num) {  return !(num&(num-1)) && !((num-1)%3);  }  C++3 (8ms)  bool isPowerOfFour(int num) {  return !(num&(num-1)) && (num & 0x55555555);  }  C++3 (3ms)  bool isPowerOfFour(int num) {  if(num<=0) return false;  while(num > 1)  {  if(num%4 != 0) return false;  num /= 4;  }  return num == 1;  }  Java 1 (2ms)  public boolean isPowerOfFour(int num) {  return (Math.log10(num)/Math.log10(4))%1 < 1e-9;  }  Java2 (22ms)  public boolean isPowerOfFour(int num) {  return Integer.toString(num, 4).matches("10\*");  } |
| 372. Super Pow  Your task is to calculate *ab* 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  C++1  class Solution {  int base = 1337;  public:  int Pow(int a, int n)  {  a %= base;  int res = 1;  for(int i=0; i<n; ++i)  res = (res\*a) % base;  return res;  }  int superPow(int a, vector<int>& b) {  if(b.empty()) return 1;  int digit = b.back();  b.pop\_back();  return (Pow(superPow(a, b), 10) \* Pow(a, digit)) % base;  }  }; |
| 50. Pow(x, n)  Implement pow(*x*, *n*).  C++1 ([**Runtime Error**](https://leetcode.com/submissions/detail/102606397/))  double myPow(double x, int n) {  if(n<0) return 1/myPow(x, -n); //n=-2147483648  if(n==0) return 1;  if(n==1) return x;  if(n==2) return x\*x;  return (n%2==0 ? 1 : x) \* myPow(myPow(x, n/2), 2);  }  C++2  double myPow(double x, int n) {  if(n == 0)  return 1;  if(n<0){  if(n == INT\_MIN) return (1/x)\*myPow(x, n+1);  n = -n; //or return myPow(x, -n)  x = 1/x;  }  return (n%2 == 0) ? myPow(x\*x, n/2) : x\*myPow(x\*x, n/2);  }  C++3  double myPow(double x, int n) {  x = (n>0) ? x:1/x;  double res=1;  for(double temp=x; n!=0; n/=2)  {  if (n%2!=0) res\*=temp;  temp\*=temp;  }  return res;  }  C++3  double myPow(double x, int n) {  if(n==0) return 1;  double t = myPow(x, n/2);  return (n%2==0 ? 1 : ((n<0)? 1/x : x)) \* t\*t;  } |
| Hr Even Odd Query You are given an array *A* of size *N*. You are also given an integer *Q*. Can you figure out the answer to each of the *Q* queries?  Each query contains 2 integers x and y, and you need to find whether the value find(x,y) is Odd or Even:  find(int x,int y)  {  if(x>y) return 1;  ans = pow(A[x],find(x+1,y))  return ans  }  Note : pow(a,b) = *ab*.  **Input Format**  The first line of the input contains an integer *N*. The next line contains *N* space separated non-negative integers(whole numbers less than or equal to 9).  The line after that contains a positive integer, *Q* , the denotes the number of queries to follow. *Q* lines follow, each line contains two positive integer *x* and *y* separated by a single space.  **Output Format**  For each query, display 'Even' if the value returned is Even, otherwise display 'Odd'.  **Constraints**  2 ≤ *N* ≤ 105  2 ≤ *Q* ≤ 105  1 ≤ *x,y* ≤ *N*  *x* ≤ *y*  Array is 1-indexed.  *No 2 consecutive entries in the array will be zero.*  **Sample Input**  3  3 2 7  2  1 2  2 3  **Sample Output**  Odd  Even  **Explanation**  find(1,2) = 9, which is Odd  find(2,3) = 128, which is even  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int N = sc.nextInt();  int[] A = new int[N];  for(int i=0; i<N; ++i)  A[i] = sc.nextInt();  int Q = sc.nextInt();  for(int i=0; i<Q; ++i){  int x = sc.nextInt();  int y = sc.nextInt();  if(x!=y && A[x]==0) System.out.println("Odd");  else if( (A[x-1]%2) == 1) System.out.println("Odd");  //if(x!=y && A[x]==0 || (A[x-1]%2) == 1) System.out.println("Odd");  else System.out.println("Even");  }  } |
| Hr Russian Peasant Exponentiation We all know how to calculate ab using b operations by multiplying 1 by a a total of b times. The drawback to this method is that b can be large, which makes exponentiation very slow.  There is a well known method called Russian Peasant Multiplication that you can read about [here](https://www.hackerrank.com/external_redirect?to=http://lafstern.org/matt/col3.pdf). Now let's use this to raise some complex numbers to powers!  You're given q queries where each query consists of four integers: a, b, k, and m. For each query, calculate (a + b·i)k = c + d·i (where i is an imaginary unit) and then print the respective values of c mod m and d mod m as two space-separated integers on a new line.  **Input Format**  The first line contains a single integer, q, denoting the number of queries.  Each of the q subsequent lines describes a query in the form of four space-separated integers: a, b, k, and m (respectively).  **Constraints**   * 1 <= q <= 105 * 0 <= k <= 1018 * 2 <= m <= 109 * 0 <= a,b <= m   **Output Format**  For each query, print the two space-separated integers denoting the respective values of c mod m and d mod m on a new line.  **Sample Input**  3  2 0 9 1000  0 1 5 10  8 2 10 1000000000  **Sample Output**  512 0  0 1  880332800 927506432  **Explanation**  In the first query, we have a=2, b=0, k=9, m=1000. We calculate the following:   1. 29 = 512 2. i5 = i   C++1  vector<long> Multiply(vector<long>& ab, vector<long>& cd, long m){  vector<long> res(2);  res[0] = (ab[0]\*cd[0]%m - ab[1]\*cd[1]%m + m) % m;  res[1] = (ab[0]\*cd[1]%m + ab[1]\*cd[0]%m + m) % m;  return res;  }  vector<long> Cal(vector<long>& ab, long k, long m){  if(k==0){  vector<long>k0(2, 0);  k0[0] = 1;  return k0;  }  if(k==1) return ab;  vector<long> res = Cal(ab, k/2, m);  res = Multiply(res, res, m);  if(k%2 == 1)  res = Multiply(res, ab, m);  return res;  }  int main() {  int q;  vector<long> ab(2);  vector<long> res(2);  long k, m;  cin>>q;  for(int i=0; i<q; ++i){  cin>>ab[0];  cin>>ab[1];  cin>>k;  cin>>m;  res = Cal(ab, k, m);  cout<<to\_string(res[0])+" "+to\_string(res[1])<<endl;  }  return 0;  }  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int q = sc.nextInt();  long [] ab = new long[2];  long k;  long m;    for(int i=0; i<q; ++i){  ab[0] = sc.nextLong();  ab[1] = sc.nextLong();  k = sc.nextLong();  m = sc.nextLong();  long[] res = Cal(ab, k, m);  System.out.println(res[0]+" "+res[1]);  }  }    private static long[] Cal(long[] ab, long k, long m){  if(k==0) return new long[]{(long)1, (long)0};  if(k==1) return ab;  long [] res = Cal(ab, k/2, m);  res = Multiply(res, res, m);  if(k%2 == 1)  res = Multiply(res, ab, m);  return res;  }    private static long[] Multiply(long[] ab, long[] cd, long m){  long[] res = new long[2];  res[0] = (ab[0]\*cd[0] % m - ab[1]\*cd[1] % m + m) % m;  res[1] = (ab[0]\*cd[1] % m + ab[1]\*cd[0] % m) % m;  return res;  } |
|  |

### Base

|  |
| --- |
|  |
| 504. Base 7  Given an integer, return its base 7 string representation.  **Example 1:**  **Input:** 100  **Output:** "202"  **Example 2:**  **Input:** -7  **Output:** "-10"  **Note:** The input will be in range of [-1e7, 1e7].  C++1 (6ms)  string convertToBase7(int num) {  if(num == 0) return "0";  int base = abs(num);  string res;  while(base != 0)  {  res = to\_string(base%7) + res;  base /= 7;  }  return (num>0) ? res : "-"+res;  }  C++2 (9ms)  class Solution {  public:  string myRescursive(int num)  {  if(num==0) return "";  return myRescursive(num/7) + to\_string(num%7);  }  string convertToBase7(int num) {  if(num==0) return "0";  return ((num>0) ? "" : "-") + myRescursive(abs(num));  }  }; |
| Hr Jim and the Jokes Jim runs a big burger restaurant and, to entertain his customers, he always tell them jokes. He is running out of jokes and he needs you to help him find new ones.  An often heard programmer joke goes like this:  "Why do programmers always mix up Christmas and Halloween? Because Dec 25 is Oct 31".  Got it? :-) It is because 2510 (25 in Decimal) is equal to 318 (31 in Octal).  If we are talking about dates, then let m be the month and d be the date and the corresponding value be f(m, d) = dm (d in base m). Let's describe some slightly different jokes:  "Why do programmers always mix up event x and event y? Because f(mx, dx) = f(my, dy) ".  Here mx means the month of event x and dx the day of event x. Similar for my and dy.  Jim knows that his customers love this kind of jokes. That's why he gives you a calendar with N events in it and asks you to count the number of such jokes he can create using the given events.  Two jokes ((x1, x2) and (y1, y2)) differ if they don't contain the same events.  **Note**:   * The given numbers are all represented with digits from 0-9, that's why for months like 11 or 12, we can't use additional characters to represent 10 or 11. * It might happen, that a special event cannot be used for a joke because the base conversion is invalid. For example 252 is not possible since base 2 can only contain digits 0 and 1. * Unary base is invalid. * Two events can have the same date.   **Input Format**  On the first line you will get N. The following N lines you will be given the dates mi, di of the special events, each separated by a single space.  **Output Format**  Print the number of jokes Jim can make.  **Constraints**   * 1 <= N <= 105 * (mi, di) will be a valid date in the Gregorian Calendar without leap day.   **Sample Input #1**  2  10 25  8 31  **Sample Output #1**  1  **Sample Input #2**  2  2 25  2 25  **Sample Output #2**  0  **Sample Input #3**  2  11 10  10 11  **Sample Output #3**  1  **Explanation**  There are two special events happening on (10, 25) and (8, 31). He can make one joke, namely the one described in the description.  In the second test case there are no valid dates we can use for our jokes since 25 is not defined for base 2.  In the third test case f(11, 10) = 1011 = 1110 = f(10, 11).  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int N = sc.nextInt();  int m, d;  HashMap<Integer, Integer> count = new HashMap();  for(int i=0; i<N; ++i){  m = sc.nextInt();  d = sc.nextInt();  int val10 = getVal(m, d);  if(val10 != 0)  count.put(val10, count.getOrDefault(val10, 0) + 1);  }  long res = 0;  for(Integer key : count.keySet()){  long val = count.get(key);  res += (val \* (val-1)) / 2;  }  System.out.println(res);  }    private static int getVal(int m, int d){  if(m < 10){  int day = d;  while(day>0){  if(day%10 >= m) return 0;  day /= 10;  }  }  int res =0, base=1;  while(d > 0){  res += (d%10) \* base;  d /= 10;  base \*= m;  }  return res;  } |
|  |
|  |

### Sqrt

|  |
| --- |
|  |
| 507. Perfect Number  We define the Perfect Number is a **positive** integer that is equal to the sum of all its **positive** divisors except itself.  Now, given an **integer** n, write a function that returns true when it is a perfect number and false when it is not.  **Example:**  **Input:** 28  **Output:** True  **Explanation:** 28 = 1 + 2 + 4 + 7 + 14  **Note:** The input number **n** will not exceed 100,000,000. (1e8)  C++1  bool checkPerfectNumber(int num) {  int sum=1;  for(int i=2; i<=sqrt(num); ++i)  {  if(num % i == 0)  {  sum += i;  if(i != num/i) sum += num/i;  }  }  return num!=1 && sum==num;  }  C++2 (3ms)  bool checkPerfectNumber(int num) {  return num==6 || num==28 || num==496 || num==8128 || num==33550336;  } |
| Codility [CountFactors](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  A positive integer D is a *factor* of a positive integer N if there exists an integer M such that N = D \* M.  For example, 6 is a factor of 24, because M = 4 satisfies the above condition (24 = 6 \* 4).  Write a function:  int solution(int N);  that, given a positive integer N, returns the number of its factors.  For example, given N = 24, the function should return 8, because 24 has 8 factors, namely 1, 2, 3, 4, 6, 8, 12, 24. There are no other factors of 24.  Assume that:   * N is an integer within the range [1..2,147,483,647].   Complexity:   * expected worst-case time complexity is O(sqrt(N)); * expected worst-case space complexity is O(1).   C++1  **int** **solution**(**int** N) {  // write your code in C++14 (g++ 6.2.0)  **if**(N==1) **return** 1;  **int** sq = **sqrt**(N);  **int** res = 2;  **for**(**int** i=2; i<=sq; ++i)  {  **if**(N%i ==0) res += 2;  }  **return** (sq\*sq == N) ? res-1 : res;  }  Testcase  squares  N=16, N=36  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  tiny  N <= 10  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  simple1  N=41(prime), N=42  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  simple2  N=69, N=64, N=120=5!  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  simple3  N=720=6!, N=1111  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  simple4  N=5,040=7!, N=12,345  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  simple5  N=34,879, N=40,320=8!  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  extreme\_one  N=1  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  medium1  N=362,880=9!, N=1,948,102  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  medium2  N=3,628,800=10!, N=5,621,892, N=4,999,696  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  big1  N=27,043,111, N=39,916,800=11!, N = 39,992,976  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  big2  N=97,093,212, N=2^28  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  big3  N=479,001,600=12!, N=780291637(prime), N=449,991,369  [▶](https://codility.com/demo/results/trainingRBFH2Q-DBT/)  extreme\_maxint  N=1,000,000,000, N=MAX\_INT, N=2147,395,600 |
| Colidity [MinPerimeterRectangle](https://codility.com/demo/results/training9H7DVY-J99/)  An integer N is given, representing the area of some rectangle.  The *area* of a rectangle whose sides are of length A and B is A \* B, and the *perimeter* is 2 \* (A + B).  The goal is to find the minimal perimeter of any rectangle whose area equals N. The sides of this rectangle should be only integers.  For example, given integer N = 30, rectangles of area 30 are:   * (1, 30), with a perimeter of 62, * (2, 15), with a perimeter of 34, * (3, 10), with a perimeter of 26, * (5, 6), with a perimeter of 22.   Write a function:  int solution(int N);  that, given an integer N, returns the minimal perimeter of any rectangle whose area is exactly equal to N.  For example, given an integer N = 30, the function should return 22, as explained above.  Assume that:   * N is an integer within the range [1..1,000,000,000].   Complexity:   * expected worst-case time complexity is O(sqrt(N)); * expected worst-case space complexity is O(1).   C++1  **int** **solution**(**int** N) {  // write your code in C++14 (g++ 6.2.0)  **int** sq = **sqrt**(N);  **for**(**int** i=sq; i>1; --i)  **if**(N%i == 0) **return** 2\*(i + N/i);  **return** 2\*(1+N);  }  Testcase  extreme\_min  N = 1 test  [▶](https://codility.com/demo/results/training9H7DVY-J99/)  simple1  N = 36 test  [▶](https://codility.com/demo/results/training9H7DVY-J99/)  simple2  N = 48 test  [▶](https://codility.com/demo/results/training9H7DVY-J99/)  simple3  N = 101 test  [▶](https://codility.com/demo/results/training9H7DVY-J99/)  small  N = 1,234 test  [▶](https://codility.com/demo/results/training9H7DVY-J99/)  medium  N = 4,564,320 test  [▶](https://codility.com/demo/results/training9H7DVY-J99/)  prime1  N = 15,486,451 test  [▶](https://codility.com/demo/results/training9H7DVY-J99/)  square  N = 100,000,000 test  [▶](https://codility.com/demo/results/training9H7DVY-J99/)  prime2  N = 982,451,653 test  [▶](https://codility.com/demo/results/training9H7DVY-J99/)  extreme\_max  N = 1,000,000,000 test |
| Codility [Flags](https://codility.com/demo/results/trainingQW92E7-9UK/)  A non-empty zero-indexed array A consisting of N integers is given.  A *peak* is an array element which is larger than its neighbours. More precisely, it is an index P such that 0 < P < N − 1 and A[P − 1] < A[P] > A[P + 1].  For example, the following array A:  A[0] = 1 A[1] = 5 A[2] = 3 A[3] = 4 A[4] = 3 A[5] = 4 A[6] = 1 A[7] = 2 A[8] = 3 A[9] = 4 A[10] = 6 A[11] = 2  has exactly four peaks: elements 1, 3, 5 and 10.  You are going on a trip to a range of mountains whose relative heights are represented by array A, as shown in a figure below. You have to choose how many flags you should take with you. The goal is to set the maximum number of flags on the peaks, according to certain rules.  https://codility-frontend-prod.s3.amazonaws.com/media/task_static/flags/static/images/auto/6f5e8faa3000c0a74157e6e0bc759b8a.png  Flags can only be set on peaks. What's more, if you take K flags, then the distance between any two flags should be greater than or equal to K. The distance between indices P and Q is the absolute value |P − Q|.  For example, given the mountain range represented by array A, above, with N = 12, if you take:   * two flags, you can set them on peaks 1 and 5; * three flags, you can set them on peaks 1, 5 and 10; * four flags, you can set only three flags, on peaks 1, 5 and 10.   You can therefore set a maximum of three flags in this case.  Write a function:  int solution(vector<int> &A);  that, given a non-empty zero-indexed array A of N integers, returns the maximum number of flags that can be set on the peaks of the array.  For example, the following array A:  A[0] = 1 A[1] = 5 A[2] = 3 A[3] = 4 A[4] = 3 A[5] = 4 A[6] = 1 A[7] = 2 A[8] = 3 A[9] = 4 A[10] = 6 A[11] = 2  the function should return 3, as explained above.  Assume that:   * N is an integer within the range [1..400,000]; * each element of array A is an integer within the range [0..1,000,000,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **vector**<**int**> peaks;  **for**(**unsigned** **int** i=1; i<A.size()-1; ++i)  {  **if**(A[i-1] < A[i] && A[i] > A[i+1])  peaks.push\_back(i);  }  **int** pSize = peaks.size();  **if**(pSize<2) **return** pSize;  **int** maxF = **sqrt**(peaks.back() - peaks[0])+1;  **int** res = 1;  **for**(**int** k=maxF; k>=2 && k>res; --k)  {  **int** lastFlag = peaks[0];  **int** used = 1;  **for**(**int** j = 1; j<pSize && used<k; ++j)  {  **if**(peaks[j] - lastFlag >= k)  {  lastFlag = peaks[j];  used++;  }  }  **if**(used > res) res = used;  }  **return** res;  }  Testcase  single  extreme min test  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  triple  three elements  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  extreme\_without\_peaks  test without peaks  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  simple1  first simple test  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  simple2  second simple test  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  medium\_many\_peaks  medium test with 100 peaks  ✘  WRONG ANSWER  got 25 expected 28  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  medium\_random  chaotic medium sequences, length = ~10,000  ✘  WRONG ANSWER  got 95 expected 97  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  packed\_peaks  possible to set floor(sqrt(N))+1 flags  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  large\_random  chaotic large sequences, length = ~100,000  ✘  WRONG ANSWER  got 315 expected 316  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  large\_little\_peaks  large test with 20-800 peaks  ✘  WRONG ANSWER  got 361 expected 437  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  large\_many\_peaks  large test with 10,000 - 25,000 peaks  ✘  WRONG ANSWER  got 309 expected 313  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  large\_anti\_slow  large test anti slow solutions  ✘  WRONG ANSWER  got 315 expected 316  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  large\_anti\_slow2  large test anti slow solutions  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  extreme\_max  extreme test, maximal number of elements  [▶](https://codility.com/demo/results/training7GQDHG-DB8/)  extreme\_max2  extreme test, maximal number of elements |
| Codility [CountNonDivisible](https://codility.com/demo/results/training3SN3RH-BEM/)  You are given a non-empty zero-indexed array A consisting of N integers.  For each number A[i] such that 0 ≤ i < N, we want to count the number of elements of the array that are not the divisors of A[i]. We say that these elements are non-divisors.  For example, consider integer N = 5 and array A such that:  A[0] = 3 A[1] = 1 A[2] = 2 A[3] = 3 A[4] = 6  For the following elements:   * A[0] = 3, the non-divisors are: 2, 6, * A[1] = 1, the non-divisors are: 3, 2, 3, 6, * A[2] = 2, the non-divisors are: 3, 3, 6, * A[3] = 3, the non-divisors are: 2, 6, * A[4] = 6, there aren't any non-divisors.   Write a function:  vector<int> solution(vector<int> &A);  that, given a non-empty zero-indexed array A consisting of N integers, returns a sequence of integers representing the amount of non-divisors.  The sequence should be returned as:   * a structure Results (in C), or * a vector of integers (in C++), or * a record Results (in Pascal), or * an array of integers (in any other programming language).   For example, given:  A[0] = 3 A[1] = 1 A[2] = 2 A[3] = 3 A[4] = 6  the function should return [2, 4, 3, 2, 0], as explained above.  Assume that:   * N is an integer within the range [1..50,000]; * each element of array A is an integer within the range [1..2 \* N].   Complexity:   * expected worst-case time complexity is O(N\*log(N)); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **vector**<**int**> solution(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **unordered\_map**<**int**, **int**> count;  **for**(**int** a : A)  count[a]++;  **int** n = A.size();  **vector**<**int**>res;  **for**(**int** a : A)  {  **int** div = (a==1) ? count[a] : count[a] + count[1];  **int** sq = **sqrt**(a);  **for**(**int** i=2; i<=sq; ++i)  {  **if**(a%i == 0)  div += count[i] + ((i\*i==a) ? 0 : count[a/i]);  }  res.push\_back(n - div);  }  **return** res;  }  Testcase  extreme\_simple  extreme simple  [▶](https://codility.com/demo/results/training3SN3RH-BEM/)  double  two elements  [▶](https://codility.com/demo/results/training3SN3RH-BEM/)  simple  simple tests  [▶](https://codility.com/demo/results/training3SN3RH-BEM/)  primes  prime numbers  [▶](https://codility.com/demo/results/training3SN3RH-BEM/)  small\_random  small, random numbers, length = 100  [▶](https://codility.com/demo/results/training3SN3RH-BEM/)  medium\_random  medium, random numbers length = 5,000  [▶](https://codility.com/demo/results/training3SN3RH-BEM/)  large\_range  1, 2, ..., N, length = ~20,000  [▶](https://codility.com/demo/results/training3SN3RH-BEM/)  large\_random  large, random numbers, length = ~30,000  [▶](https://codility.com/demo/results/training3SN3RH-BEM/)  large\_extreme  large, all the same values, length = 50,000 |
| Hr Best Divisor Kristen loves playing with and comparing numbers. She thinks that if she takes two different positive numbers, the one whose digits sum to a larger number is *better* than the other. If the sum of digits is equal for both numbers, then she thinks the smaller number is *better*. For example, Kristen thinks that 13 is better than 31 and that 12 is better than 11.  Given an integer, n, can you find the divisor of n that Kristin will consider to be the best?  **Input Format**  A single integer denoting n.  **Constraints**   * 0 <= n <= 105   **Output Format**  Print an integer denoting the best divisor of n.  **Sample Input 0**  12  **Sample Output 0**  6  **Explanation 0**  The set of divisors of 12 can be expressed as {1,2,3,4,6,12}. The divisor whose digits sum to the largest number is 6 (which, having only one digit, sums to itself). Thus, we print 6 as our answer.  Java1  public static void main(String[] args) {  Scanner in = new Scanner(System.in);  int n = in.nextInt();  int res = 1;  int s = (int)Math.sqrt(n);  for(int i=1; i<=s; ++i){  if(n%i == 0) {  res = compare(res, i);  if(n/i != i)  res = compare(res, n/i);  }  }  System.out.println(res);  }    private static int compare(int a, int b){  int sumA = SumDigit(a);  int sumB = SumDigit(b);  if(sumA==sumB) return Math.min(a, b);  return sumA>sumB ? a : b;  }    private static int SumDigit(int x) {  int res = 0;  while(x>0){  res += x%10;  x /= 10;  }  return res;  } |
| Hr Sherlock and Divisors Watson gives an integer N to Sherlock and asks him: What is the number of divisors of N that are divisible by 2?.  **Input Format**  First line contains T, the number of testcases. This is followed by T lines each containing an integer N.  **Output Format**  For each testcase, print the required answer in one line.  **Constraints**   1 <= T <= 100 1 <= N <=109  **Sample Input**  2  9  8  **Sample Output**  0  3  **Explanation**  9 has three divisors 1, 3 and 9 none of which is divisible by 2.  8 has four divisors 1,2,4 and 8, out of which three are divisible by 2.  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  int N;  for(int i=0; i<T; ++i){  N = sc.nextInt();  System.out.println(CountDivisors(N));  }  }    private static int CountDivisors(int n){  int s = (int)Math.sqrt(n);  int res = 0;  for(int i=1; i<=s; ++i){  if(n%i == 0){  if(i%2==0) res++;  if(n/i != i && (n/i)%2 == 0) res++;  }  }  return res;  } |
| HrBus Station There are  **groups of friends**, and each group is numbered from 1 to n. The ith group contains  people.  They live near a bus stop, and only a single bus operates on this route. An empty bus arrives at the bus stop and all the groups want to travel by the bus.  However, **group of friends** do not want to get separated. So they enter the bus only if the bus can carry the entire group.  Moreover, the groups do not want to change their relative positioning while travelling. In other words, group 3 cannot travel by bus, unless group 1 and group 2 have either (a) already traveled by the bus in the previous trip or (b) they are also sitting inside the bus at present.  You are given that a bus of size x can carry x people simultaneously.   Find the size x of the bus so that (1) the bus can transport all the groups and (2) every time when the bus starts from the bus station, there is no empty space in the bus (i.e. the total number of people present inside the bus is equal to x)?  **Input Format**  The first line contains an integer n (1 <= n <= 105) . The second line contains n space-separated integers a1, a2,…, an (1 <= ai <= 104) .  **Output Format**  Print all possible sizes of the bus in an increasing order.  **Sample Input**  8  1 2 1 1 1 2 1 3  **Sample Output**  3 4 6 12  **Sample Explanation**  In the above example, a1 = 1, a2 = 2, a3 = 1, a4 = 1, a5 = 1, a6 = 2, a7 = 1, a8 = 3.  If x = 1 : In the first trip, a1 go by the bus. There will be no second trip because the bus cannot accommodate group 2. Hence "x = 1" is not the required answer.  If x = 2 : No bus trip is possible. That's because a1 cannot go alone, as one seat will be left vacant in the bus. And, a1 & a2 cannot go together, because the bus is cannot accommodate both the groups simultaneously.  If x = 3 : In the first trip, a1 & a2 go by the bus. In the second trip, a3, a4 & a5 go by the bus. In the third trip, a6 & a7 go by the bus. In the fourth trip, a8 go by the bus.  If x = 4 : In the first trip, a1, a2 & a3 go by the bus. In the second trip, a4, a5 & a6 go by the bus. In the third trip,  a7& a8 go by the bus.  Similarly you can figure out the output for x= 5, 6 & 7.  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int n = sc.nextInt();  Set<Integer> sumSet = new HashSet<Integer>();  List<Integer> res = new ArrayList<Integer>();  int sum = 0;  for(int i=0; i<n; ++i){  sum += sc.nextInt();  sumSet.add(sum);  }  int s = (int)Math.sqrt(sum);  for(int i=1; i<=s; ++i){  if(sum % i != 0) continue;  if(IsPossibleSize(i, sum, sumSet))  res.add(i);  if(sum/i != i && IsPossibleSize(sum/i, sum, sumSet))  res.add(sum/i);  }  Collections.sort(res);  for(int i=0; i<res.size(); ++i)  System.out.print(res.get(i) + " ");  System.out.println();  }    private static boolean IsPossibleSize(int size, int sum, Set<Integer> sumSet){  int sumS = size;  while(sumS < sum){  if(!sumSet.contains(sumS))  return false;  sumS += size;  }  return true;  } |
| 633. Sum of Square Numbers  Given a non-negative integer c, your task is to decide whether there're two integers a and b such that a2 + b2 = c.  **Example 1:**  **Input:** 5  **Output:** True  **Explanation:** 1 \* 1 + 2 \* 2 = 5  **Example 2:**  **Input:** 3  **Output:** False  Java1  public boolean judgeSquareSum(int c) {  int n = (int)Math.sqrt(c);  for(int a=0; a<=n; ++a){  int b2 = c - a\*a;  int b = (int)Math.sqrt(b2);  if(b\*b == b2)  return true;  }  return false;  } |
|  |

### Odd and Even

|  |
| --- |
| Hr Halloween party Alex is attending a Halloween party with his girlfriend, Silvia. At the party, Silvia spots the corner of an infinite chocolate bar (two dimensional, infinitely long in width and length).  If the chocolate can be served only as 1 x 1 sized pieces and Alex can cut the chocolate bar exactly K times, what is the maximum number of chocolate pieces Alex can cut and give Silvia?  **Input Format**  The first line contains an integer T, the number of test cases. T lines follow. Each line contains an integer K.  **Output Format** T lines; each line should contain an integer that denotes the maximum number of pieces that can be obtained for each test case.  **Constraints** 1 <= T <= 10 2 <= K <= 107  **Note**: Chocolate must be served in *1 x 1* sized pieces. Alex can't relocate any of the pieces, nor can he place any piece on top of another.  **Sample Input #00**  4  5  6  7  8  **Sample Output #00**  6  9  12  16  **Explanation** The explanation below is for the first two test cases. The rest of them follow a similar logic.  For the first test-case where K=5, you need 3 horizontal and 2 vertical cuts. halloweenboard  For the second test case, where K=6, you need 3 horizontal and 3 vertical cuts.  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  for(int i=0; i<T; ++i){  long K = sc.nextLong();  System.out.println(K%2==0 ? (K/2)\*(K/2) : (K/2)\*(K/2 +1));  }  } |
|  |
|  |

## 计算几何(Computational Geometry)

线段相交,

多边形面积,

内点外点的判断,

凸包(Convex Hull),

重心(Bary Center)...

### 线性代数

#### 矩阵(Matrix)

|  |
| --- |
| 463. 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:  https://leetcode.com/static/images/problemset/island.png  C++1  int islandPerimeter(vector<vector<int>>& grid) {  int row = grid.size();  int col = (row > 0) ? grid[0].size() : 0;  int res = 0;  for(int i=0; i<row; ++i)  {  for(int j=0; j<col; ++j)  {  if(grid[i][j] == 1)  {  if(i==0 || grid[i-1][j]==0 ) res++;  if(j==0 || grid[i][j-1]==0) res++;  if(i+1==row || grid[i+1][j]==0) res++;  if(j+1==col || grid[i][j+1]==0) res++;  }  }  }  return res;  }  Python1  def islandPerimeter(self, grid):  res=0; h=len(grid); w=len(grid[0]) if h>0 else 0  for i in range(h):  for j in range(w):  if grid[i][j] == 1 :  if i-1<0 or grid[i-1][j]==0:  res += 1  if j-1<0 or grid[i][j-1]==0:  res += 1  if i+1==h or grid[i+1][j]==0:  res += 1  if j+1==w or grid[i][j+1]==0:  res += 1  return res |
| 74. 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.  C++1  bool searchMatrix(vector<vector<int>>& matrix, int target) {  int h = matrix.size();  int w = (h>0) ? matrix[0].size() : 0;  int i=0, j= h\*w - 1;  while(i<=j)  {  int mid = i + (j-i)/2;  int ele = matrix[mid/w][mid%w];  if(ele == target) return true;  else if(ele > target) j = mid-1;  else i = mid+1;  }  return false;  } |
| 240. 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.  C++1 (59ms)  bool searchMatrix(vector<vector<int>>& matrix, int target) {  int h = matrix.size();  int w = (h>0) ? matrix[0].size() : 0;  int i= 0, j = w-1;  while(i<h && j>=0)  {  if(target == matrix[i][j]) return true;  else if(target > matrix[i][j]) i++;  else j--;  }  return false;  } |
| 542. 01 Matrix  Given a matrix consists of 0 and 1, find the distance of the nearest 0 for each cell.  The distance between two adjacent cells is 1.  **Example 1:** Input:  0 0 0  0 1 0  0 0 0  Output:  0 0 0  0 1 0  0 0 0  **Example 2:** Input:  0 0 0  0 1 0  1 1 1  Output:  0 0 0  0 1 0  1 2 1  **Note:**   1. The number of elements of the given matrix will not exceed 10,000. 2. There are at least one 0 in the given matrix. 3. The cells are adjacent in only four directions: up, down, left and right.   C++1 (316ms)  class Solution {  public:  void updateFromTo(int h, int w, vector<vector<int>>& matrix, vector<vector<int>>& res, bool frontToEnd)  {  for(int i = (frontToEnd) ? 0 : h-1; (frontToEnd) ? i<h : i>=0; (frontToEnd) ? ++i : --i)  for(int j = (frontToEnd) ? 0 : w-1; (frontToEnd) ? j<w : j>=0; (frontToEnd) ? ++j : --j)  {  if(matrix[i][j] == 0) res[i][j] = 0;  else  {  if(i>0) res[i][j] = min(res[i][j], res[i-1][j]+1);  if(j>0) res[i][j] = min(res[i][j], res[i][j-1]+1);  if(i+1<h) res[i][j] = min(res[i][j], res[i+1][j]+1);  if(j+1<w) res[i][j] = min(res[i][j], res[i][j+1]+1);  }  }  }  vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {  int h = matrix.size(), w=matrix[0].size();  vector<vector<int>> res(h, vector<int>(w, 10000));  updateFromTo(h,w,matrix,res, true);  updateFromTo(h,w,matrix,res, false);  return res;  }  }; |
| 48. 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?  C++1 (6ms)  void rotate(vector<vector<int>>& matrix) {  int n = matrix.size()-1;  for(int i=0; i<n; ++i)  {  for(int j=i; j<n-i; ++j)  {  int temp = matrix[i][j];  matrix[i][j] = matrix[n-j][i];  matrix[n-j][i] = matrix[n-i][n-j];  matrix[n-i][n-j] = matrix[j][n-i];  matrix[j][n-i] = temp;  }  }  }  C++2 (4ms)  void rotate(vector<vector<int>>& matrix) {  int n = matrix.size()-1;  for(int i=0; i<(n+1)/2; ++i)  {  for(int j=i; j<n-i; ++j)  {  swap(matrix[j][i], matrix[i][n-j]);  swap(matrix[n-i][j], matrix[j][i]);  swap(matrix[n-j][n-i], matrix[n-i][j]);  }  }  }  Java1 (1ms)  public class Solution {  public void rotate(int[][] matrix) {  int n = matrix.length-1;  for(int i=0; i<(n+1)/2; ++i)  {  for(int j=i; j<n-i; ++j)  {  //up to left  mySwap(matrix, i,n-j, j,i);  //left to down  mySwap(matrix, j,i, n-i,j);  //down to right  mySwap(matrix, n-i,j, n-j,n-i);  }  }  }    private void mySwap(int[][] matrix, int ai, int aj, int bi, int bj)  {  int temp = matrix[ai][aj];  matrix[ai][aj] = matrix[bi][bj];  matrix[bi][bj] = temp;  }  } |
|  |
| 566. Reshape the Matrix  In MATLAB, there is a very useful function called 'reshape', which can reshape a matrix into a new one with different size but keep its original data.  You're given a matrix represented by a two-dimensional array, and two **positive** integers **r** and **c** representing the **row** number and **column** number of the wanted reshaped matrix, respectively.  The reshaped matrix need to be filled with all the elements of the original matrix in the same **row-traversing** order as they were.  If the 'reshape' operation with given parameters is possible and legal, output the new reshaped matrix; Otherwise, output the original matrix.  **Example 1:**  **Input:**  nums =  [[1,2],  [3,4]]  r = 1, c = 4  **Output:**  [[1,2,3,4]]  **Explanation:** The **row-traversing** of nums is [1,2,3,4]. The new reshaped matrix is an 1 \* 4 matrix, fill it row by row by using the previous list.  **Example 2:**  **Input:**  nums =  [[1,2],  [3,4]]  r = 2, c = 4  **Output:**  [[1,2],  [3,4]]  **Explanation:** There is no way to reshape a 2 \* 2 matrix to a 2 \* 4 matrix. So output the original matrix.  **Note:**   1. The height and width of the given matrix is in range [1, 100]. 2. The given r and c are all positive.   C++1  vector<vector<int>> matrixReshape(vector<vector<int>>& nums, int r, int c) {  int h=nums.size(), w=nums[0].size();  if(r\*c != h\*w) return nums;  vector<vector<int>> res(r, vector<int>(c));  for(int i=0, k=0; i<r; ++i)  for(int j=0; j<c; ++j, ++k)  res[i][j] = nums[k/w][k%w];  return res;  } |
| 221. Maximal Square  Given a 2D binary matrix filled with 0's and 1's, find the largest square containing only 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.  C++1  int maximalSquare(vector<vector<char>>& matrix) {  int h = matrix.size();  int w = h>0 ? matrix[0].size() : 0;  vector<int> dp(w+1);  int edge=0, topLeft=0;  for(int i=0; i<h; ++i)  {  for(int j=1; j<=w; ++j)  {  int temp = dp[j];  if(matrix[i][j-1]=='1')  {  dp[j] = min(dp[j], min(dp[j-1], topLeft)) + 1;  edge = max(edge, dp[j]);  }  else  dp[j] = 0;  topLeft = temp;  }  }  return edge\*edge;  } |
| 54. 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].  C++1  vector<int> spiralOrder(vector<vector<int>>& matrix) {  int h=matrix.size();  int w = h>0 ? matrix[0].size() : 0;  int size = h\*w;  vector<int> res(size);  if(size==0) return res;  vector<vector<bool>> vis(h, vector<bool>(w));  vis[0][0]=true;  res[0]=matrix[0][0];  int n=0,i=0,j=0;  while(n<size-1)  {  while(j+1<w && !vis[i][j+1]) { vis[i][j+1]=true; res[++n]=matrix[i][++j];}  while(i+1<h && !vis[i+1][j]) { vis[i+1][j]=true; res[++n]=matrix[++i][j];}  while(j-1>=0 && !vis[i][j-1]) { vis[i][j-1]=true; res[++n]=matrix[i][--j];}  while(i-1>=0 && !vis[i-1][j]) { vis[i-1][j]=true; res[++n]=matrix[--i][j];}  }  return res;  }  C++2  vector<int> spiralOrder(vector<vector<int>>& matrix) {  if(matrix.empty()) return vector<int>();  vector<int> res(1, matrix[0][0]);  int m = matrix.size(), n = matrix[0].size();  vector<vector<bool>> visited(m, vector<bool>(n, false));  visited[0][0]=true;  int i=0, j=0, loop = (min(m,n)+1)/2;  for(int k=0; k<loop; ++k)  {  while(j+1<n && !visited[i][j+1])  {  visited[i][j+1]=true;  res.push\_back(matrix[i][++j]);  }  while(i+1<m && !visited[i+1][j])  {  visited[i+1][j]= true;  res.push\_back(matrix[++i][j]);  }  while(j-1>=0 && !visited[i][j-1])  {  visited[i][j-1]=true;  res.push\_back(matrix[i][--j]);  }  while(i-1>=0 && !visited[i-1][j])  {  visited[i-1][j]=true;  res.push\_back(matrix[--i][j]);  }  }  return res;  }  C++3  vector<int> spiralOrder(vector<vector<int>>& matrix) {  if(matrix.empty()) return vector<int>();    int m = matrix.size(), n = matrix[0].size();  vector<int> res(m\*n);  int left=0, right=n-1, up=0, down=m-1, i=0;  while(true)  {  for(int col=left; col<=right; ++col) res[i++] = matrix[up][col];  if(++up>down) break;    for(int row=up; row<=down; ++row) res[i++] = matrix[row][right];  if(--right<left) break;    for(int col=right; col>=left; --col) res[i++] = matrix[down][col];  if(--down<up) break;    for(int row=down; row>=up; --row) res[i++] = matrix[row][left];  if(++left>right) break;  }  return res;  } |
| 598. Range Addition II  Given an m \* n matrix **M** initialized with all **0**'s and several update operations.  Operations are represented by a 2D array, and each operation is represented by an array with two **positive** integers **a** and **b**, which means **M[i][j]** should be **added by one** for all **0 <= i < a** and **0 <= j < b**.  You need to count and return the number of maximum integers in the matrix after performing all the operations.  **Example 1:**  **Input:**  m = 3, n = 3  operations = [[2,2],[3,3]]  **Output:** 4  **Explanation:**  Initially, M =  [[0, 0, 0],  [0, 0, 0],  [0, 0, 0]]  After performing [2,2], M =  [[1, 1, 0],  [1, 1, 0],  [0, 0, 0]]  After performing [3,3], M =  [[2, 2, 1],  [2, 2, 1],  [1, 1, 1]]  So the maximum integer in M is 2, and there are four of it in M. So return 4.  **Note:**   1. The range of m and n is [1,40000]. 2. The range of a is [1,m], and the range of b is [1,n]. 3. The range of operations size won't exceed 10,000.   C++1  int maxCount(int m, int n, vector<vector<int>>& ops) {  int a = m, b = n;  for(auto op : ops)  {  a = min(a, op[0]);  b = min(b, op[1]);  }  return a\*b;  } |
| Hr Matrix Tracing A word from the English dictionary is taken and arranged as a matrix. e.g. "MATHEMATICS"  MATHE  ATHEM  THEMA  HEMAT  EMATI  MATIC  ATICS  There are many ways to trace this matrix in a way that helps you construct this word. You start tracing the matrix from the top-left position and at each iteration, you either move RIGHT or DOWN, and ultimately reach the bottom-right of the matrix. It is assured that any such tracing generates the same word. How many such tracings can be possible for a given word of length m+n-1 written as a matrix of size m \* n?  **Input Format**  The first line of input contains an integer T. T test cases follow.  Each test case contains 2 space separated integers m & n (in a new line) indicating that the matrix has m rows and each row has n characters.  **Constraints**  1 <= T <= 103  1 ≤ m,n ≤ 106  **Output Format**  Print the number of ways (S) the word can be traced as explained in the problem statement. If the number is larger than 109+7,  print S mod (10^9 + 7) for each testcase (in a new line).  **Sample Input**  1  2 3  **Sample Output**  3  **Explanation**  Let's consider a word AWAY written as the matrix  AWA  WAY  Here, the word AWAY can be traced in 3 different ways, traversing either RIGHT or DOWN.  AWA  Y  AW  AY  A  WAY  Hence the answer is 3.  Java1 (MLE)  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  int m, n;  for(int i=0; i<T; ++i){  m = sc.nextInt();  n = sc.nextInt();  System.out.println(CountPaths(m, n));  }  }    private static int CountPaths(int h, int w){  if(h < w) return CountPaths(w, h);  int [] dp = new int[w];  Arrays.fill(dp, 1);  for(int i=1; i<h; ++i){  for(int j=1; j<w; ++j)  dp[j] = (dp[j]+dp[j-1]) % 1000000007;  }  return dp[w-1];  } |
| Hr Bot saves princess Princess Peach is trapped in one of the four corners of a square grid. You are in the center of the grid and can move one step at a time in any of the four directions. Can you rescue the princess?  **Input format**  The first line contains an odd integer N (3 <= N < 100) denoting the size of the grid. This is followed by an NxN grid. Each cell is denoted by '-' (ascii value: 45). The bot position is denoted by 'm' and the princess position is denoted by 'p'.  Grid is indexed using [Matrix Convention](https://www.hackerrank.com/scoring/board-convention)  **Output format**  Print out the moves you will take to rescue the princess in one go. The moves must be separated by '\n', a newline. The valid moves are LEFT or RIGHT or UP or DOWN.  **Sample input**  3  ---  -m-  p--  **Sample output**  DOWN  LEFT  **Task**  Complete the function *displayPathtoPrincess* which takes in two parameters - the integer N and the character array grid. The grid will be formatted exactly as you see it in the input, so for the sample input the princess is at grid[2][0]. The function shall output moves (LEFT, RIGHT, UP or DOWN) on consecutive lines to rescue/reach the princess. The goal is to reach the princess in as few moves as possible.  The above sample input is just to help you understand the format. The princess ('p') can be in any one of the four corners.  **Scoring**  Your score is calculated as follows : (NxN - number of moves made to rescue the princess)/10, where N is the size of the grid (3x3 in the sample testcase).  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int N = sc.nextInt();  int si=0, sj=0, ei=0, ej=0;  for(int i=0; i<N; ++i){  String line = sc.next();  char[] c = line.toCharArray();  for(int j=0; j<N; ++j){  if(c[j] == 'm'){  si = i;  sj = j;  }  else if(c[j] == 'p'){  ei = i;  ej = j;  }  }  }  while(si!=ei || sj!=ej){  if(si<ei){  si++;  System.out.println("DOWN");  }  else if(si>ei){  si--;  System.out.println("UP");  }    if(sj < ej){  sj++;  System.out.println("RIGHT");  }  else if(sj > ej){  sj--;  System.out.println("LEFT");  }  }  } |
| 130. Surrounded Regions  Given a 2D board containing 'X' and 'O' (the **letter** 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  C++1  void DFS(vector<vector<char>>& board, int i, int j)  {  board[i][j] = 'S';  if(i-1>0 && board[i-1][j] == 'O') DFS(board, i-1, j);  if(i+1<board.size() && board[i+1][j] == 'O') DFS(board, i+1, j);  if(j-1>0 && board[i][j-1] == 'O') DFS(board, i, j-1);  if(j+1<board[0].size() && board[i][j+1] == 'O') DFS(board, i, j+1);  }  void solve(vector<vector<char>>& board) {  int h = board.size();  int w = h>0 ? board[0].size() : 0;  for(int i=0; i<h; ++i)  {  if(board[i][0] == 'O') DFS(board, i, 0);  if(board[i][w-1] == 'O') DFS(board, i, w-1);  }  for(int i=1; i<w-1; ++i)  {  if(board[0][i] == 'O') DFS(board, 0, i);  if(board[h-1][i] == 'O') DFS(board, h-1, i);  }  for(int i=0; i<h; ++i)  for(int j=0; j<w; ++j)  if(board[i][j] == 'S') board[i][j] = 'O';  else board[i][j] = 'X';  } |
| # LC999. Available Captures for Rook  ### LeetCode  ## Question  On an 8 x 8 chessboard, there is one white rook. There also may be empty squares, white bishops, and black pawns. These are given as characters 'R', '.', 'B', and 'p' respectively. Uppercase characters represent white pieces, and lowercase characters represent black pieces.  The rook moves as in the rules of Chess: it chooses one of four cardinal directions (north, east, west, and south), then moves in that direction until it chooses to stop, reaches the edge of the board, or captures an opposite colored pawn by moving to the same square it occupies. Also, rooks cannot move into the same square as other friendly bishops.  Return the number of pawns the rook can capture in one move.    \*\*Example 1:\*\*  ![LC999. Available Captures for Rook](Images/LC999AvailableCapturesForRook1.png)  ```  Input: [[".",".",".",".",".",".",".","."],[".",".",".","p",".",".",".","."],[".",".",".","R",".",".",".","p"],[".",".",".",".",".",".",".","."],[".",".",".",".",".",".",".","."],[".",".",".","p",".",".",".","."],[".",".",".",".",".",".",".","."],[".",".",".",".",".",".",".","."]]  Output: 3  Explanation:  In this example the rook is able to capture all the pawns.  ```  \*\*Example 2:\*\*  ![LC999. Available Captures for Rook](Images/LC999AvailableCapturesForRook2.png)  ```  Input: [[".",".",".",".",".",".",".","."],[".","p","p","p","p","p",".","."],[".","p","p","B","p","p",".","."],[".","p","B","R","B","p",".","."],[".","p","p","B","p","p",".","."],[".","p","p","p","p","p",".","."],[".",".",".",".",".",".",".","."],[".",".",".",".",".",".",".","."]]  Output: 0  Explanation:  Bishops are blocking the rook to capture any pawn.  ```  \*\*Example 3:\*\*  ![LC999. Available Captures for Rook](Images/LC999AvailableCapturesForRook3.png)  ```  Input: [[".",".",".",".",".",".",".","."],[".",".",".","p",".",".",".","."],[".",".",".","p",".",".",".","."],["p","p",".","R",".","p","B","."],[".",".",".",".",".",".",".","."],[".",".",".","B",".",".",".","."],[".",".",".","p",".",".",".","."],[".",".",".",".",".",".",".","."]]  Output: 3  Explanation:  The rook can capture the pawns at positions b5, d6 and f5.  ```  \*\*Note:\*\*  \* board.length == board[i].length == 8  \* board[i][j] is either 'R', '.', 'B', or 'p'  \* There is exactly one cell with board[i][j] == 'R'  ## Solutions  ### Solution 1  \* Java  ```  class Solution {  public int numRookCaptures(char[][] board) {  int[] R = findRook(board);  int pawns = 0;  for(int i=R[0]; i>=0; i--){  if(board[i][R[1]] == 'B') break;  else if(board[i][R[1]] == 'p'){  pawns++;  break;  }  }  for(int i=R[0]; i<board.length; i++){  if(board[i][R[1]] == 'B') break;  else if(board[i][R[1]] == 'p'){  pawns++;  break;  }  }  for(int i=R[1]; i>=0; i--){  if(board[R[0]][i] == 'B') break;  else if(board[R[0]][i] == 'p'){  pawns++;  break;  }  }  for(int i=R[1]; i<board.length; i++){  if(board[R[0]][i] == 'B') break;  else if(board[R[0]][i] == 'p'){  pawns++;  break;  }  }  return pawns;  }    private int[] findRook(char[][] board){  for(int i=0; i<board.length; i++)  for(int j=0; j<board[0].length; j++)  if(board[i][j] == 'R')  return new int[]{i,j};  return new int[2];  }  }  ```  The code looks too long. I have not got a solution to clean it up.  \* \*\*worst-case time complexity:\*\* `O(m \* n)`, where `m` is the row of the `board`, n is the height of the `board`.  \* \*\*worst-case space complexity:\*\* `O(1)` |
|  |

#### 线性方程组(Linear Equations)

|  |
| --- |
| 441. 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.  C++1 (15ms)  int arrangeCoins(int n) {  long coin = sqrt(n);  while((coin\*(coin+1))/2 <=n)  coin++;  return coin-1;  }  C++2 (12ms)  int arrangeCoins(int n) {  long layer = sqrt(2\*(long)n);  int sum = (layer \* (layer+1)) / 2;  return (sum > n) ? layer-1 : layer;  }  C++3 (9ms)  int arrangeCoins(int n) {  long sum = 2\*(long)n;  long layer = sqrt(sum);  long res = (layer \* (layer+1))/2;  return (res > n) ? layer-1 : layer;  }  C++4 (9ms)  1+2+3+...+x = n -> (1+x)x/2 = n -> x^2+x = 2n -> x^2+x+1/4 = 2n +1/4 -> (x+1/2)^2 = 2n +1/4 -> (x+0.5) = sqrt(2n+0.25) -> x = -0.5 + sqrt(2n+0.25)  int arrangeCoins(int n) {  return floor(-0.5+sqrt((double)2\*n+0.25)); }  C++ (log(n))  int arrangeCoins(int n) {  long low = 1, high = n;  while (low < high) {  long mid = low + (high - low + 1) / 2;  if ((mid + 1) \* mid / 2.0 <= n) low = mid;  else high = mid - 1;  }  return high;  } |
| 168. 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  C++1(0ms)  string convertToTitle(int n) {  string res = "";  while(n>0)  {  res = (char)('A' - 1 + ((n%26 == 0)? 26: n%26)) + res;  n = n/26 - ((n%26 == 0)? 1 : 0);  }  return res;  }  C++2 (0ms)  string convertToTitle(int n) {  if(n==0) return "";  return convertToTitle((n-1)/26) + (char)('A' + (n-1)%26);  }  C++3  string convertToTitle(int n) {  string res = "";  while(n>0)  {  res = (char)('A' + (n-1)%26) + res;  n = (n-1)/26;  }  return res;  }  Java1  public String convertToTitle(int n) {  StringBuilder sb = new StringBuilder();  while(n>0)  {  sb.insert(0, (char)('A' +(n-1)%26));  n = (n-1) / 26;  }  return sb.toString();  } |
| 400. Nth Digit  Find the *n*th 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* < 231).  **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.  C++1 (3ms)  int findNthDigit(int n) {  int len=1, start =1;  long count = 9;  while(n > len\*count)  {  n -= len\*count; //minus the length of a kind of number (1digit, 2 digits,….)  len++; // (1digit, 2 digits,….)  count \*= 10; //the count of the same digit number (9, 90 ,900, …)  start \*= 10; //the start number of the same digit number (1, 10, 100…)  }  start += (n-1)/len;  string res = to\_string(start);  return res[(n-1)%len] - '0';  } |
| 357. Count Numbers with Unique Digits  Given a **non-negative** integer n, count all numbers with unique digits, x, where 0 ≤ x < 10n.  **Example:** Given n = 2, return 91. (The answer should be the total numbers in the range of 0 ≤ x < 100, excluding [11,22,33,44,55,66,77,88,99])  **Hint:**   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 10n. 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].   C++1 (0ms)  int countNumbersWithUniqueDigits(int n) {  n = (n>10) ? 10 : n;  int res = 1, dig = 9, cur = 9;  for(int i=1; i<=n; ++i)  {  res += cur;  cur \*= dig--;  }  return res;  }  C++2  int countNumbersWithUniqueDigits(int n) {  int dp[11] = {1};  dp[1] = 10;  n = min(10, n);  for(int i=2; i<=n; ++i)  {  dp[i] = 9;  int temp = 9;  for(int j=1; j<i; ++j)  dp[i] \*= temp--;  dp[i] += dp[i-1];  }  return dp[n];  }  Java1  public int countNumbersWithUniqueDigits(int n) {  int[] dp = new int[11];  dp[0] = 1;  dp[1] = 10;  n = Math.min(n, 10);  for(int i=2; i<=n; ++i)  {  dp[i] = 9;  int temp = 9;  for(int j=1; j<i; ++j)  dp[i] \*= temp--;  dp[i] += dp[i-1];  }  return dp[n];  } |
| 492. 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.   C++1 (3ms) O()  vector<int> constructRectangle(int area) {  vector<int>res(2);  int w = sqrt(area);  for(int i=w; i>0; --i)  {  int L = area/i;  if(L\*i == area)  {  res[0] = L;  res[1] = i;  return res;  }  }  return res;  } |
| 62. Unique Paths  A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).  The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).  How many possible unique paths are there?  http://leetcode.com/wp-content/uploads/2014/12/robot_maze.png  Above is a 3 x 7 grid. How many possible unique paths are there?  **Note:** *m* and *n* will be at most 100.  C++1 (0ms) O(mn)  int uniquePaths(int m, int n) {  if(n<1) return 0;  vector<int> dp(n, 1);  for(int i=1; i<m; ++i)  for(int j=1; j<n; ++j)  dp[j] += dp[j-1];  return dp[n-1];  }  C++2  int uniquePaths(int m, int n) {  if(m>n) return uniquePaths(n, m);  double res=1;  for(int i=m-1; i>0; --i)  {  res \*= (double)(n-1+i)/i;  }  return round(res);  }  Java1  public int uniquePaths(int m, int n) {  if(m < n)  return uniquePaths(n, m);  if( n<= 0) return 0;  int[] dp = new int[n];  Arrays.fill(dp, 1);  for(int i=1; i<m; ++i)  for(int j=1; j<n; ++j)  dp[j] += dp[j-1];  return dp[n-1];  } |
| 390. Elimination Game  There is a list of sorted integers from 1 to *n*. Starting from left to right, remove the first number and every other number afterward until you reach the end of the list.  Repeat the previous step again, but this time from right to left, remove the right most number and every other number from the remaining numbers.  We keep repeating the steps again, alternating left to right and right to left, until a single number remains.  Find the last number that remains starting with a list of length *n*.  **Example:**  Input:  n = 9,  1 2 3 4 5 6 7 8 9  2 4 6 8  2 6  6  Output:  6  C++1  int lastRemaining(int n) {  if(n<1) return 0;  int start = 1, step = 1, remaining = n;  bool leftToRight = true;  while(remaining > 1)  {  if(leftToRight || remaining%2 == 1)  start += step;  remaining /= 2;  step \*= 2;  leftToRight = !leftToRight;  }  return start;  }  C++2  int lastRemaining(int n) {  return (n==1)? 1 : 2\*(1 + n/2 - lastRemaining(n/2));  } |
| 292. 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.  **Hint:**   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?   C++1  bool canWinNim(int n) {  return n%4!=0;  }  Python1  def canWinNim(self, n):  return bool(n&3) |
| 258. 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?  **Hint:**   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](https://en.wikipedia.org/wiki/Digital_root) useful.   C++1  int addDigits(int num) {  return (num!=0 && num%9==0) ? 9 : num%9;  } |
| 268. 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?  C++1 (36ms)  int missingNumber(vector<int>& nums) {  int len = nums.size();  int sumN = len\*(len+1)/2;  for(int n : nums) sumN -= n;  return sumN;  }  Java1 (1ms)  public int missingNumber(int[] nums) {  int n = nums.length;  int sum = 0;  for(int num : nums)  sum += num;  return n\*(n+1)/2 - sum;  } |
| 202. 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   * 12 + 92 = 82 * 82 + 22 = 68 * 62 + 82 = 100 * 12 + 02 + 02 = 1   C++1 (3ms)  bool isHappy(int n) {  while(n>6)  {  int sum = 0;  while(n>0)  {  int digit = n%10;  n/=10;  sum += digit\*digit;  }  n = sum;  }  return n==1;  } |
| 537. Complex Number Multiplication  Given two strings representing two [complex numbers](https://en.wikipedia.org/wiki/Complex_number).  You need to return a string representing their multiplication. Note i2 = -1 according to the definition.  **Example 1:**  **Input:** "1+1i", "1+1i"  **Output:** "0+2i"  **Explanation:** (1 + i) \* (1 + i) = 1 + i2 + 2 \* i = 2i, and you need convert it to the form of 0+2i.  **Example 2:**  **Input:** "1+-1i", "1+-1i"  **Output:** "0+-2i"  **Explanation:** (1 - i) \* (1 - i) = 1 + i2 - 2 \* i = -2i, and you need convert it to the form of 0+-2i.  **Note:**   1. The input strings will not have extra blank. 2. The input strings will be given in the form of **a+bi**, where the integer **a** and **b** will both belong to the range of [-100, 100]. And **the output should be also in this form**.   C++1 (3ms)  class Solution {  public:  vector<int> getNums(string s)  {  int i=0;  vector<int> res(2);  while(s[i]!='+') ++i;  res[0] = stoi(s.substr(0, i));  res[1] = stoi(s.substr(i+1, s.length()-1));  return res;  }  string complexNumberMultiply(string a, string b) {  vector<int> A = getNums(a);  vector<int> B = getNums(b);  return to\_string(A[0]\*B[0] - A[1]\*B[1]) + "+" + to\_string(A[0]\*B[1] + A[1]\*B[0]) + "i";  }  };  C++2  vector<int> getNums(string s)  {  int i=0;  while(s[i]!='+') ++i;  int x = stoi(s.substr(0, i));  int y = stoi(s.substr(i+1, s.length()-1));  return vector<int> ({x, y});  } |
| Codility [FrogJmp](https://codility.com/demo/results/training7G27XE-FF3/#task-0)  A small frog wants to get to the other side of the road. The frog is currently located at position X and wants to get to a position greater than or equal to Y. The small frog always jumps a fixed distance, D.  Count the minimal number of jumps that the small frog must perform to reach its target.  Write a function:  int solution(int X, int Y, int D);  that, given three integers X, Y and D, returns the minimal number of jumps from position X to a position equal to or greater than Y.  For example, given:  X = 10 Y = 85 D = 30  the function should return 3, because the frog will be positioned as follows:   * after the first jump, at position 10 + 30 = 40 * after the second jump, at position 10 + 30 + 30 = 70 * after the third jump, at position 10 + 30 + 30 + 30 = 100   Assume that:   * X, Y and D are integers within the range [1..1,000,000,000]; * X ≤ Y.   Complexity:   * expected worst-case time complexity is O(1); * expected worst-case space complexity is O(1).   C++1  **int** **solution**(**int** X, **int** Y, **int** D) {  // write your code in C++14 (g++ 6.2.0)  **int** s = (Y-X)/D;  **return** (s\*D == (Y-X)) ? s : s+1;  }  TestCases  simple1  simple test  [▶](https://codility.com/demo/results/training7G27XE-FF3/)  simple2  [▶](https://codility.com/demo/results/training7G27XE-FF3/)  extreme\_position  no jump needed  [▶](https://codility.com/demo/results/training7G27XE-FF3/)  small\_extreme\_jump  one big jump  [▶](https://codility.com/demo/results/training7G27XE-FF3/)  many\_jump1  many jumps, D = 2  [▶](https://codility.com/demo/results/training7G27XE-FF3/)  many\_jump2  many jumps, D = 99  [▶](https://codility.com/demo/results/training7G27XE-FF3/)  many\_jump3  many jumps, D = 1283  [▶](https://codility.com/demo/results/training7G27XE-FF3/)  big\_extreme\_jump  maximal number of jumps  [▶](https://codility.com/demo/results/training7G27XE-FF3/)  small\_jumps  many small jumps |
| Codility [CountDiv](https://codility.com/demo/results/trainingZHQUCB-SVS/)  given three integers A, B and K, returns the number of integers within the range [A..B] that are divisible by K, i.e.:  { i : A ≤ i ≤ B, i mod K = 0 }  For example, for A = 6, B = 11 and K = 2, your function should return 3, because there are three numbers divisible by 2 within the range [6..11], namely 6, 8 and 10.  Assume that:   * A and B are integers within the range [0..2,000,000,000]; * K is an integer within the range [1..2,000,000,000]; * A ≤ B.   Complexity:   * expected worst-case time complexity is O(1); * expected worst-case space complexity is O(1).   Java  public int solution(int A, int B, int K) {  // write your code in Java SE 8  A = Math.max(A, K);  if(K==1) return B-A+1;  int res = 0;  for(int i=A; i<=B; ++i){  if(i % K == 0) res++;  }  return res;  }  C++1  **int** **solution**(**int** A, **int** B, **int** K) {  **int** res = B/K - A/K;  **return** (A%K==0) ? res+1 : res;  }  Testcase  simple  A = 11, B = 345, K = 17  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  minimal  A = B in {0,1}, K = 11  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  extreme\_ifempty  A = 10, B = 10, K in {5,7,20}  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  extreme\_endpoints  verify handling of range endpoints, multiple runs  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  big\_values  A = 100, B=123M+, K=2  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  big\_values2  A = 101, B = 123M+, K = 10K  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  big\_values3  A = 0, B = MAXINT, K in {1,MAXINT}  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  big\_values4  A, B, K in {1,MAXINT} |
| HackerRank Summing the N series    C++1  int SN(long double N){  //long double x = N\*(N+1)/2;  //long double y = N\*(N-1)/2;  long double x = fmod(N, 1e9 + 7);  return (int) fmod(x\*x, 1e9 + 7);  }  int main() {  /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/  int T;  long double N;  cin>>T;  for(int i=0; i<T; ++i){  cin>>N;  cout<< SN(N)<<endl;  }  return 0;  }  Java1  public static void main(String[] args) {  /\* Enter your code here. Read input from STDIN. Print output to STDOUT. Your class should be named Solution. \*/  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  BigInteger N;  for(int i=0; i<T; ++i){  N = sc.nextBigInteger();  System.out.println(SN(N));  }  }    private static BigInteger MOD = new BigInteger("1000000007");    private static BigInteger SN(BigInteger N){  N = N.mod(MOD);  return (N.multiply(N)).mod(MOD);  }  **Explanation:**  This is a formula solving question.  Firstly, let’s solve the formula  Second, bring the into the series  It is sum of odds from 1 to (2\*n-1). Since n could be very large as 1016, we can image how slow the sum from 1 to (2\*n-1) is.  Then, I started to think how to make the series continuous, so that I can use formula to get sum. I found that we could separate the into two parts:  Therefore the equals:  So, the answer of this question is to calculate  Since the input N is 1<=N<=1016, we could use BigInteger of Java to handle it. Before doing N multiply N, we could mod 1e9+7 first. It could decrease the result N\*N, because of (A\*B)%C = (A%C) \* (B%C). |
|  |

### 概率论

1006 Weird Clock

1008 How many N 20131005

|  |
| --- |
| 198. 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**.  C++1 (3ms)  int rob(vector<int>& nums) {  int odd=0, even=0;  for(int i=0; i<nums.size(); ++i)  {  if(i%2 == 0)  odd = max(even, odd+nums[i]);  else  even = max(odd, even+nums[i]);  }  return max(odd, even);  }  Java1 (1ms)  public int rob(int[] nums) {  int robber1=0, robber2=0;  for(int i=0; i<nums.length; ++i)  {  if(i%2==0) robber1 = Math.max(robber2, robber1+nums[i]);  else robber2 = Math.max(robber1, robber2+nums[i]);  }  return Math.max(robber1, robber2);  } |
| 213. House Robber II  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**.  C++1  class Solution {  public:  int robHelper(vector<int>& nums, int start, int end)  {  int cur, pre=0, bpre = 0;  for(int i=start; i<=end; ++i)  {  cur = max(pre, bpre+nums[i]);  bpre = pre;  pre = cur;  }  return max(pre, bpre);  }  int rob(vector<int>& nums) {  int n = nums.size();  if(n<2) return n? nums[0]:0;  return max(robHelper(nums, 0, n-2), robHelper(nums, 1, n-1));  }  };  C++2  class Solution {  public:  int maxRob(vector<int>& nums, int start, int end)  {  if(start>end) return 0;  int prepre=0, pre=nums[start];  for(int i=start+1; i<end; ++i)  {  int cur = prepre+nums[i];  prepre = pre;  pre = max(pre, cur);  }  return max(pre, prepre);  }  int rob(vector<int>& nums) {  return max(maxRob(nums, 0, nums.size()-1), maxRob(nums, 1, nums.size()));  }  }; |
| 337. 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**.  C++1 (16ms)  class Solution {  public:  vector<int> robBT(TreeNode\* root)  {  vector<int> curNext(2);  if(root==NULL) return curNext;  vector<int> left = robBT(root->left);  vector<int> right = robBT(root->right);  curNext[0] = root->val + left[1] + right[1]; //rob current layer  curNext[1] = max(left[0], left[1]) + max(right[0], right[1]);  return curNext;  }  int rob(TreeNode\* root) {  vector<int>res = robBT(root);  return max(res[0], res[1]);  }  };  Java1 (1ms)  public class Solution {  public int rob(TreeNode root) {  int[] rebSum = robTree(root);  return Math.max(rebSum[0], rebSum[1]);  }    private int[] robTree(TreeNode root)  {  int[] robSum = new int[2];  if(root==null) return robSum;  int[] robLeft = robTree(root.left);  int[] robRight = robTree(root.right);  robSum[0] = Math.max(robLeft[0], robLeft[1])  + Math.max(robRight[0], robRight[1]);  robSum[1] = root.val + robLeft[0] + robRight[0];  return robSum;  }  } |
| 169. Majority Element  Given an array of size *n*, find the majority element. The majority element is the element that appears **more than** ⌊ n/2 ⌋ times.  You may assume that the array is non-empty and the majority element always exist in the array.  C++1 (22ms)  int majorityElement(vector<int>& nums) {  int count=0, res;  for(int i=0; i<nums.size(); ++i)  {  if(count==0) res = nums[i];  if(nums[i] == res) count++;  else count--;  }  return res;  }  C++2 (44ms)  int majorityElement(vector<int>& nums) {  sort(nums.begin(), nums.end());  return nums[nums.size()/2];  }  C++3 (179ms)  int majorityElement(vector<int>& nums) {  nth\_element(nums.begin(), nums.begin()+nums.size()/2, nums.end());  return nums[nums.size()/2];  }  Java1 (3ms)  public int majorityElement(int[] nums) {  Arrays.sort(nums);  return nums[nums.length/2];  }  Java2 (44ms)  public int majorityElement(int[] nums) {  int half = nums.length/2;  Map<Integer, Integer> count = new HashMap();  for(int n : nums)  {  if(!count.containsKey(n)) count.put(n, 1);  else count.put(n, count.get(n) + 1);  if(count.get(n)>half) return n;  }  return 0;  } |
| 229. 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.  C++1  vector<int> majorityElement(vector<int>& nums) {  int ele1=1, ele2=2, count1=0, count2=0;  for(int num : nums)  {  if(num == ele1) count1++;  else if(num == ele2) count2++;  else if(count1 == 0)  {  ele1 = num;  count1++;  }  else if(count2 == 0)  {  ele2 = num;  count2++;  }  else  {  count1--;  count2--;  }  }  count1=count2=0;  for(int num : nums)  {  if(num == ele1) count1++;  else if(num==ele2) count2++;  }  vector<int> res;  if(count1 > nums.size()/3) res.push\_back(ele1);  if(count2 > nums.size()/3) res.push\_back(ele2);  return res;  } |
| Codility [Dominator](https://codility.com/demo/results/trainingCSMNGG-ASB/)  A zero-indexed array A consisting of N integers is given. The *dominator*of array A is the value that occurs in more than half of the elements of A.  For example, consider array A such that  A[0] = 3 A[1] = 4 A[2] = 3 A[3] = 2 A[4] = 3 A[5] = -1 A[6] = 3 A[7] = 3  The dominator of A is 3 because it occurs in 5 out of 8 elements of A (namely in those with indices 0, 2, 4, 6 and 7) and 5 is more than a half of 8.  Write a function  int solution(vector<int> &A);  that, given a zero-indexed array A consisting of N integers, returns index of any element of array A in which the dominator of A occurs. The function should return −1 if array A does not have a dominator.  Assume that:   * N is an integer within the range [0..100,000]; * each element of array A is an integer within the range [−2,147,483,648..2,147,483,647].   For example, given array A such that  A[0] = 3 A[1] = 4 A[2] = 3 A[3] = 2 A[4] = 3 A[5] = -1 A[6] = 3 A[7] = 3  the function may return 0, 2, 4, 6 or 7, as explained above.  Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** leader = -1, count=0;  **for**(**int** a : A)  {  **if**(count==0)  {  leader = a;  count++;  }  **else** **if**(leader == a) count++;  **else** count--;  }  count = 0;  **for**(**int** a : A)  **if**(a == leader) count++;  **if**(count <= (**int**)A.size()/2) **return** -1;    **int** res = 0;  **while**(A[res]!=leader) res++;  **return** res;  }  Testcase  small\_nondominator  all different and all the same elements  [▶](https://codility.com/demo/results/trainingCSMNGG-ASB/)  small\_half\_positions  half elements the same, and half + 1 elements the same  [▶](https://codility.com/demo/results/trainingCSMNGG-ASB/)  small  small test  [▶](https://codility.com/demo/results/trainingCSMNGG-ASB/)  small\_pyramid  decreasing and plateau, small  [▶](https://codility.com/demo/results/trainingCSMNGG-ASB/)  extreme\_empty\_and\_single\_item  empty and single element arrays  [▶](https://codility.com/demo/results/trainingCSMNGG-ASB/)  extreme\_half1  array with exactly N/2 values 1, N even + [0,0,1,1,1]  [▶](https://codility.com/demo/results/trainingCSMNGG-ASB/)  extreme\_half2  array with exactly floor(N/2) values 1, N odd + [0,0,1,1,1]  [▶](https://codility.com/demo/results/trainingCSMNGG-ASB/)  extreme\_half3  array with exactly ceil(N/2) values 1 + [0,0,1,1,1] |
| Codility [EquiLeader](https://codility.com/demo/results/trainingYJK8H4-AVF/)  A non-empty zero-indexed array A consisting of N integers is given.  The *leader* of this array is the value that occurs in more than half of the elements of A.  An *equi leader* is an index S such that 0 ≤ S < N − 1 and two sequences A[0], A[1], ..., A[S] and A[S + 1], A[S + 2], ..., A[N − 1] have leaders of the same value.  For example, given array A such that:  A[0] = 4 A[1] = 3 A[2] = 4 A[3] = 4 A[4] = 4 A[5] = 2  we can find two equi leaders:   * 0, because sequences: (4) and (3, 4, 4, 4, 2) have the same leader, whose value is 4. * 2, because sequences: (4, 3, 4) and (4, 4, 2) have the same leader, whose value is 4.   The goal is to count the number of equi leaders.  Write a function:  int solution(vector<int> &A);  that, given a non-empty zero-indexed array A consisting of N integers, returns the number of equi leaders.  For example, given:  A[0] = 4 A[1] = 3 A[2] = 4 A[3] = 4 A[4] = 4 A[5] = 2  the function should return 2, as explained above.  Assume that:   * N is an integer within the range [1..100,000]; * each element of array A is an integer within the range [−1,000,000,000..1,000,000,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** leader=0, count=0;  **for**(**int** a : A)  {  **if**(count==0)  {  leader = a;  count++;  }  **else** **if**(leader==a) count++;  **else** count--;  }    **int** n = A.size();  **vector**<**bool**> LR(n);  **vector**<**bool**> RL(n);  **int** lr =0, rl = 0;  **for**(**int** i=0; i<n-1; ++i)  {  **if**(A[i]==leader) lr++;  **if**(A[n-i-1]==leader) rl++;  **int** half = (i+1)/2;  **if**(lr>half) LR[i] = **true**;  **if**(rl>half) RL[n-i-1] = **true**;  }    **int** res = 0;  **for**(**int** i=0; i<n-1; ++i)  **if**(LR[i] && RL[i+1])  res++;  **return** res;  }  C++2  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** leader=0, count=0;  **for**(**int** a : A)  {  **if**(count==0)  {  leader = a;  count++;  }  **else** **if**(leader==a) count++;  **else** count--;  }    **int** n = A.size()-1;  **vector**<**bool**> LR(n);  **vector**<**bool**> RL(n);  **int** lr =0, rl = 0;  **for**(**int** i=0; i<n; ++i)  {  **if**(A[i]==leader) lr++;  **if**(A[n-i]==leader) rl++;  **int** half = (i+1)/2;  **if**(lr>half) LR[i] = **true**;  **if**(rl>half) RL[n-i-1] = **true**;  }    **int** res = 0;  **for**(**int** i=0; i<n; ++i)  **if**(LR[i] && RL[i])  res++;  **return** res;  }  Testcase  single  single element  [▶](https://codility.com/demo/results/trainingYJK8H4-AVF/)  double  two elements  [▶](https://codility.com/demo/results/trainingYJK8H4-AVF/)  simple  simple test  [▶](https://codility.com/demo/results/trainingYJK8H4-AVF/)  small\_random  small random test with two values, length = ~100  [▶](https://codility.com/demo/results/trainingYJK8H4-AVF/)  small  random + 200 \* [MIN\_INT] + random ,length = ~300  [▶](https://codility.com/demo/results/trainingYJK8H4-AVF/)  large\_random  large random test with two values, length = ~50,000  [▶](https://codility.com/demo/results/trainingYJK8H4-AVF/)  large  random(0,1) + 50000 \* [0] + random(0, 1), length = ~100,000  [▶](https://codility.com/demo/results/trainingYJK8H4-AVF/)  large\_range  1, 2, ..., N, length = ~100,000  [▶](https://codility.com/demo/results/trainingYJK8H4-AVF/)  extreme\_large  all the same values |
|  |

#### Random

|  |
| --- |
| 384. 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();  /\*\*  \* 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();  \*/  C++1 (232ms)  class Solution {  private:  vector<int> ori;  public:  Solution(vector<int> nums) {  ori = nums;  }    /\*\* Resets the array to its original configuration and return it. \*/  vector<int> reset() {  return ori;  }    /\*\* Returns a random shuffling of the array. \*/  vector<int> shuffle() {  vector<int> res = ori;  for(int i=1; i<res.size(); ++i)  swap(res[i], res[rand()%(i+1)]);  return res;  }  };  C++2  class Solution {  private:  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;  }  };  Java1  public class Solution {    private int[] oriArray;  private Random rand;  private int size;    public Solution(int[] nums) {  oriArray = nums;  size = nums.length;  rand = new Random();  }    /\*\* Resets the array to its original configuration and return it. \*/  public int[] reset() {  return oriArray;  }    /\*\* Returns a random shuffling of the array. \*/  public int[] shuffle() {  int[] shufArray = Arrays.copyOf(oriArray, size);  for(int i=1; i<size; ++i)  {  int r = rand.nextInt(i+1);  int temp = shufArray[i];  shufArray[i] = shufArray[r];  shufArray[r] = temp;  }  return shufArray;  }  }  Java2  public class Solution {    private int[] oriArray;  private Random rand;  private int size;    public Solution(int[] nums) {  oriArray = nums;  size = nums.length;  rand = new Random();  }    /\*\* Resets the array to its original configuration and return it. \*/  public int[] reset() {  return oriArray;  }    /\*\* Returns a random shuffling of the array. \*/  public int[] shuffle() {  int[] shufArray = Arrays.copyOf(oriArray, size);  for(int i=0; i<size; ++i)  {  int r = rand.nextInt(size - i);  int temp = shufArray[i];  shufArray[i] = shufArray[i+r];  shufArray[i+r] = temp;  }  return shufArray;  }  } |
| 382. 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 equal probability of returning.  solution.getRandom();  /\*\*  \* Definition for singly-linked list.  \* struct ListNode {  \* int val;  \* ListNode \*next;  \* ListNode(int x) : val(x), next(NULL) {}  \* };  \* Your Solution object will be instantiated and called as such:  \* Solution obj = new Solution(head);  \* int param\_1 = obj.getRandom();  \*/  C++1 (49ms)  class Solution {  private:  ListNode\* list;  public:  /\*\* @param head The linked list's head.  Note that the head is guaranteed to be not null, so it contains at least one node. \*/  Solution(ListNode\* head) {  list = head;  }    /\*\* Returns a random node's value. \*/  int getRandom() {  int res = list->val;  ListNode\* cur = list;  for(int i=1; cur!=NULL; ++i)  {  if(rand()%i == 0) res = cur->val;  cur = cur->next;  }  return res;  }  };  C++2  class Solution {  private:  vector<int> values;  int size;  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) {  while(head!=NULL)  {  values.push\_back(head->val);  head = head->next;  }  size = values.size();  srand(time(NULL));  }    /\*\* Returns a random node's value. \*/  int getRandom() {  return values[rand()%size];  }  };  Java1  public class Solution {  private ListNode myHead;  private Random rand;    /\*\* @param head The linked list's head.  Note that the head is guaranteed to be not null, so it contains at least one node. \*/  public Solution(ListNode head) {  myHead = head;  rand = new Random();  }    /\*\* Returns a random node's value. \*/  public int getRandom() {  ListNode node = myHead;  ListNode res = node;  for (int i=1; node != null; ++i)  {  if(rand.nextInt(i) == 0) //if((int)(Math.random()\*i) == 0)  res = node;  node = node.next;  }  return res.val;  }  }  Java2  public class Solution {  private List<Integer> values;  private int size;  private Random random;  /\*\* @param head The linked list's head. Note that the head is guanranteed to be not null, so it contains at least one node. \*/  public Solution(ListNode head) {  values = new ArrayList<Integer>();  while(head!=null)  {  values.add(head.val);  head = head.next;  }  size = values.size();  random = new Random();  }    /\*\* Returns a random node's value. \*/  public int getRandom() {  return values.get(random.nextInt(size));  }  } |
| 398. 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 equal 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);  C++1 (146ms)  class Solution {  private:  vector<int> myNums;  public:  Solution(vector<int> nums) {  myNums = nums;  }    int pick(int target) {  int res = -1, count=0;  for(int i=0; i<myNums.size(); ++i)  if(myNums[i] == target && (rand()% ++count == 0))  res = i;  return res;  }  }; |
|  |

#### 排列组合（[permutation and combination](http://dict.youdao.com/w/permutation%20and%20combination/#keyfrom=E2Ctranslation)）

|  |
| --- |
|  |
|  |
|  |

### 初等数学与解析几何

|  |
| --- |
| 413. Arithmetic Slices  A sequence of number 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 sequence:  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 slice of that array is any pair of integers (P, Q) such that 0 <= P < Q < N.  A slice (P, Q) of array A is called arithmetic if the sequence: A[P], A[p + 1], ..., A[Q - 1], A[Q] is arithmetic. In particular, this means that P + 1 < Q.  The function should return the number of arithmetic slices in the array A.    **Example:**  A = [1, 2, 3, 4]  return: 3, for 3 arithmetic slices in A: [1, 2, 3], [2, 3, 4] and [1, 2, 3, 4] itself.  C++1 (3ms) O(n)  int numberOfArithmeticSlices(vector<int>& A) {  int res =0, ALen = A.size(), count=0;  vector<int> minus(ALen);  for(int i=1; i<ALen; ++i)  minus[i] = A[i] - A[i-1];  for(int i=2; i<ALen; ++i)  {  if(minus[i] == minus[i-1])  {  count++;  continue;  }  if(count>0)  {  res += (count\*(count+1))/2;  count = 0;  }  }  return (count>0) ? res + (count\*(count+1))/2 : res;  }  Java1 (2ms) O(n)  public int numberOfArithmeticSlices(int[] A) {  int ALen = A.length, res=0, count=0;  int[] minus = new int[ALen];  for(int i=1; i<ALen; ++i)  minus[i] = A[i]-A[i-1];  for(int i=2; i<ALen; ++i)  {  if(minus[i] == minus[i-1])  count++;  else if(count>0)  {  res += (count\*(count+1))/2;  count = 0;  }  }  return count>0 ? res + count\*(count+1)/2 : res;  } |
| 447. 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 iand 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]]**  C++1 (1000ms)  int numberOfBoomerangs(vector<pair<int, int>>& points) {  int booms = 0;  for (auto &p : points) {  unordered\_map<double, int> ctr(points.size());  for (auto &q : points)  //hypot is the  *square root* of (x2+y2),distance  booms += 2 \* ctr[hypot(p.first - q.first, p.second - q.second)]++;  }  return booms;  }  Java1 (282ms)  public class Solution {  public int numberOfBoomerangs(int[][] points) {  int res = 0;  for(int i=0; i<points.length; ++i)  {  Map<Integer, Integer> groups = new HashMap<>();  for(int j=0; j<points.length; ++j)  {  int dis = DistanceOf(points[i], points[j]);  if(!groups.containsKey(dis)) groups.put(dis, 1);  else groups.put(dis, groups.get(dis)+1);  }  for(Integer key : groups.keySet())  {  int size = groups.get(key);  res += size \*(size-1);  }  }  return res;  }  private int DistanceOf(int[] point1, int[] point2)  {  int a = point1[0] - point2[0];  int b = point1[1] - point2[1];  return a\*a + b\*b;  }  } |
| 223. 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.  Rectangle Area  Assume that the total area is never beyond the maximum possible value of **int**.  C++1  int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {  int area1 = (C-A) \* (D-B);  int area2 = (G-E) \* (H-F);  if(E>=C || F>=D || G<=A || H<=B) return area1+area2;  int shareArea = (min(D,H)-max(B,F))\*(min(G,C)-max(A,E));  return area1 + area2 - shareArea;  }  C++2  int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {  int areas = (C-A) \* (D-B) + (G-E) \* (H-F);  if(E>=C || F>=D || G<=A || H<=B) return areas;  return areas - (min(D,H)-max(B,F))\*(min(G,C)-max(A,E));  } |
| 593. Valid Square  Given the coordinates of four points in 2D space, return whether the four points could construct a square.  The coordinate (x,y) of a point is represented by an integer array with two integers.  **Example:**  **Input:** p1 = [0,0], p2 = [1,1], p3 = [1,0], p4 = [0,1]  **Output:** True  Note:   1. All the input integers are in the range [-10000, 10000]. 2. A valid square has four equal sides with positive length and four equal angles (90-degree angles). 3. Input points have no order.   C++1 (could solve range more than 10000)  class Solution {  public:  double lenFrom(vector<int>& p1, vector<int>& p2)  {  double a = p1[0] - p2[0];  double b = p1[1] - p2[1];  return a\*a + b\*b;  }  bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {  vector<double> lens(6);  lens[0] = lenFrom(p1, p2);  lens[1] = lenFrom(p1, p3);  lens[2] = lenFrom(p1, p4);  lens[3] = lenFrom(p2, p3);  lens[4] = lenFrom(p2, p4);  lens[5] = lenFrom(p3, p4);  sort(lens.begin(), lens.end());  for(int i=0; i<3; ++i)  if(abs(lens[i]-lens[i+1]) > 1e-9)  return false;  if(abs(lens[4]-lens[5]) > 1e-9) return false;  if(abs(lens[0]-lens[5]) < 1e-9) return false;  if(abs(lens[0]-lens[5]/2) > 1e-9) return false;  return true;  }  };  C++2  int d(vector<int>& p1, vector<int>& p2) {  return (p1[0] - p2[0]) \* (p1[0] - p2[0]) + (p1[1] - p2[1]) \* (p1[1] - p2[1]);  }  bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {  unordered\_set<int> s({ d(p1, p2), d(p1, p3), d(p1, p4), d(p2, p3), d(p2, p4), d(p3, p4) });  return !s.count(0) && s.size() == 2;  } |
| Find the Point (HackerRank) Consider two points, P = (Px, Py) and Q = (Qx, Qy). We consider the inversion or [point reflection](https://www.hackerrank.com/external_redirect?to=https://en.wikipedia.org/wiki/Point_reflection), R = (Rx, Ry) , of point P across point Q to be a 180o rotation of point P around Q.  Given  sets of points P and Q, find R for each pair of points and print two space-separated integers denoting the respective values of Rx and Ry on a new line.  **Input Format**  The first line contains an integer, , denoting the number of sets of points.  Each of the  subsequent lines contains four space-separated integers describing the respective values of Px, Py, Qx, Qy defining points P = (Px, Py) and Q = (Qx, Qy).  **Constraints**   * 1<= n <= 15 * -100 <= Px, Py, Qx, Qy <= 100   **Output Format**  For each pair of points  and , print the corresponding respective values of  and  as two space-separated integers on a new line.  **Sample Input**  2  0 0 1 1  1 1 2 2  **Sample Output**  2 2  3 3  **Explanation**  The graphs below depict points P, Q, and R for the n=2  points given as Sample Input:   1. find-point-0011.png Thus, we print  Rx and Ry  as 2 2 on a new line. 2. find-point-1122.png Thus, we print Rx and Ry  as 3 3 on a new line.   C++1  int main() {  /\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/  int n;  int p[4];  cin >> n;  for(int i=0; i<n; ++i) {  for(int j=0; j<4; ++j)  cin >> p[j];  cout << to\_string(p[2]\*2 - p[0]) + " " + to\_string(p[3]\*2 - p[1]) << endl;  }  return 0;  }  Java1  public static void main(String[] args) {  /\* Enter your code here. Read input from STDIN. Print output to STDOUT. Your class should be named Solution. \*/  Scanner sc = new Scanner(System.in);  int n = sc.nextInt();  int [] p = new int[4];  for(int i=0; i<n && sc.hasNextInt(); ++i) {  for(int j=0; j<4; ++j)  p[j] = sc.nextInt();  int x = p[2]\*2 - p[0];  int y = p[3]\*2 - p[1];  System.out.println(x+" "+y);  }  } |
| Hr Sherlock and Moving Tiles Sherlock is given 2 square tiles, initially both of whose sides have length L placed in an x-y plane; so that the bottom left corner of each square coincides with the the origin and their sides are parallel to the axes.  At t=0, both squares start moving along line y=x (along the positive x and y) with velocities S1 and S2.  For each query of form qi, Sherlock has to report the time at which the overlapping area of tiles is equal to qi.    **Note**: Assume all distances in meter, time in seconds and velocities in meter per second unless otherwise specified.  **Input Format**  First line contains integers L, S1, S2. Next line contains Q, the number of queries. Each of the next Q lines consists of one integer qi in one line.  **Constraints**   1 <= L, S1, S2 <= 109  1 <= Q <= 105  1 <= qi <= L2  **Output Format**  For each query, print the required answer in one line. Your answer will be considered correct if it is at most 0.0001 away from the true answer. See the explanation for more details.  **Sample Input**  10 1 2  2  50  100  **Sample Output**  4.1421  0.0000  **Explanation**  For the first case, note that the answer is around 4.1421356237..., so any of the following will be accepted:  4.1421356237  4.14214  4.14215000  4.1421  4.1422  Java1  private static final double SQRT2 = Math.sqrt(2);  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  double L = sc.nextDouble();  double S1 = sc.nextDouble();  double S = Math.abs(sc.nextDouble() - S1);  int Q = sc.nextInt();    for(int i=0; i<Q; ++i){  double q = sc.nextDouble();  double res = (L-Math.sqrt(q))\*SQRT2 / S;  System.out.printf("%.4f\n", res);  }  }  Explanation |
| Hr Most Distant Keko has N dots in a 2-D coordinate plane. He wants to measure the gap between the most distant two dots. To make the problem easier, Keko decided to change each dot's x *or* y coordinate to zero.  Help Keko calculate the distance!  **Input Format**  The first line contains an integer, N, the number of dots.  The next N lines each contain the integer coordinates of the dots in (x,y) fashion.  **Constraints**   2 <= N <= 106  -109 <= xi, yi <= 109  It is guaranteed that all dots are distinct, and either their x or y coordinate is equal to 0.  **Output Format**  Print the distance between the most distant dots with an absolute error of, at most, 10-6.  **Sample Input**  4  -1 0  1 0  0 1  0 -1  **Sample Output**  2.000000  **Explanation**  In the sample, the most distant dots are located at (-1,0) and (1,0).  The distance between them is 2.  C++1  double Distance(double p1[2], double p2[2]){  double x = p1[0] - p2[0];  double y = p1[1] - p2[1];  return sqrt(x\*x + y\*y);  }  int main() {  int N;  double x, y;  cin>>N;  double p[4][2] = {0};  for(int i=0; i<N; ++i){  cin>>x>>y;  p[0][0] = max(p[0][0], x);  p[1][0] = min(p[1][0], x);  p[2][1] = max(p[2][1], y);  p[3][1] = min(p[3][1], y);  }  double res = 0.0;  for(int i=0; i<3; ++i)  for(int j=i+1; j<4; ++j)  res = max(res, Distance(p[i], p[j]));  //cout<<res<<endl;  printf("%.12f\n", res);  return 0;  }  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int N = sc.nextInt();  double x, y;  double[][] p = new double[4][2];  for(int i=0; i<N; ++i){  x = sc.nextDouble();  y = sc.nextDouble();  p[0][0] = Math.max(p[0][0], x);  p[1][0] = Math.min(p[1][0], x);  p[2][1] = Math.max(p[2][1], y);  p[3][1] = Math.min(p[3][1], y);  }  double res = 0.0;  for(int i=0; i<3; ++i){  for(int j=i+1; j<4; ++j)  res = Math.max(res, Distance(p[i], p[j]));  }  System.out.printf("%.12f\n", res);  }    private static double Distance(double[] p1, double[] p2){  double x = p1[0] - p2[0];  double y = p1[1] - p2[1];  return Math.sqrt(x\*x + y\*y);  } |
|  |

1010 Clock Hour and Minute Hands Angle 20131013

### 高等数学

#### 点积(Dot Product)

#### 差积(Cross Product)

#### 积分(Integral)

#### 微分(Differential)

## Foundation

|  |
| --- |
| HR Maximum Draws Jim is off to a party and is searching for a matching pair of socks. His drawer is filled with socks, each pair of a different color. In its worst case scenario, how many socks (x) should Jim remove from his drawer until he finds a matching pair?  **Input Format**  The first line contains the number of test cases T.  Next T lines contains an integer N which indicates the total pairs of socks present in the drawer.  **Output Format**  Print the number of Draws (x) Jim makes in the worst case scenario.  **Constraints**   1 <= T <= 1000 0 < N < 106  **Sample Input**  2  1  2  **Sample Output**  2  3  **Explanation**  Case 1 : A pair of socks are present, hence exactly 2 draws for the socks to match.  Case 2 : 2 pair of socks are present in the drawer. The first and the second draw might result in 2 socks of different color. The 3rd sock picked will definitely match one of previously picked socks. Hence, 3.  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  for(int i=0; i<T; ++i){  System.out.println(sc.nextInt()+1);  }  } |
| Handshake At the annual meeting of Board of Directors of Acme Inc, every one starts shaking hands with everyone else in the room. Given the fact that any two persons shake hand exactly once, Can you tell the total count of handshakes?  **Input Format**  The first line contains the number of test cases T, T lines follow.  Each line then contains an integer N, the total number of Board of Directors of Acme.  **Output Format**  Print the number of handshakes for each test-case in a new line.  **Constraints**  1 <= T <= 1000  0 < N < 106  **Sample Input**  2  1  2  **Sample Output**  0  1  **Explanation**  Case 1 : The lonely board member shakes no hands, hence 0.  Case 2 : There are 2 board members, 1 handshake takes place.  Java1  public static void main(String[] args) {  Scanner in = new Scanner(System.in);  int T = in.nextInt();  for(int a0 = 0; a0 < T; a0++){  int N = in.nextInt();  System.out.println((long)N\*(N-1)/2);  }  } |
| Hr Army Game Luke is daydreaming in Math class. He has a sheet of graph paper with n rows and m columns, and he imagines that there is an army base in each cell for a total of n · m bases. He wants to drop supplies at strategic points on the sheet, marking each drop point with a red dot. If a base contains at least one package inside or on top of its border fence, then it's considered to be supplied. For example:  image  Given n and m, what's the minimum number of packages that Luke must drop to supply all of his bases?  **Input Format**  Two space-separated integers describing the respective values of n and m.  **Constraints**  0 < n,m <= 1000  **Output Format**  Print a single integer denoting the minimum number of supply packages Luke must drop.  **Sample Input 0**  2 2  **Sample Output 0**  1  **Explanation 0**  Luke has four bases in a 2 X 2 grid. If he drops a single package where the walls of all four bases intersect, then those four cells can access the package:  image  Because he managed to supply all four bases with a single supply drop, we print 1 as our answer.  Java1  public static void main(String[] args) {  Scanner in = new Scanner(System.in);  int n = in.nextInt();  int m = in.nextInt();  System.out.println(((m+1)/2) \* ((n+1)/2));  } |
| HR Leonardo's Prime Factors Leonardo loves primes and created q queries where each query takes the form of an integer, n. For each n, he wants you to count the maximum number of unique prime factors of any number in the inclusive range [1,n] and then print this value on a new line.  **Note:** Recall that a prime number is only divisible by 1 and itself, and 1 is *not* a prime number.  **Input Format**  The first line contains an integer, q, denoting the number of queries.  Each line i of the j subsequent lines contains a single integer, n.  **Constraints**   * 1 <= q <= 105 * 1 <= n <= 1018   **Output Format**  For each query, print the maximum number of unique prime factors for any number in the inclusive range [1,n] on a new line.  **Sample Input**  6  1  2  3  500  5000  10000000000  **Sample Output**  0  1  1  4  5  10  **Explanation**   1. The maximum number of unique prime factors of any number in the inclusive range [1,1] is 0, because 1 is not prime and its only factor is itself. 2. The maximum number of unique prime factors of any number in the inclusive range [1,2] is 1. We already know that the number 1 has 0 prime factors, but 2 has 1 prime factor (itself). 3. The maximum number of unique prime factors of any number in the inclusive range [1,3] is 1. The number 3 has 1 prime factor (itself), and we already know that the number 2 has 1 prime factor and the number 1 has 0 prime factors. 4. The maximum number of unique prime factors in the inclusive range [1,500] is 4. The product of our first four unique primes is 2×3×5×7=210, and there are no additional unique primes we can multiply that number by that results in a value <=500.   Java1  public static void main(String[] args) {  String[] primes = new String[]  {"1","2","3","5","7","11","13","17","19","23","29","31","37","41","43","47","53"};  BigInteger[] sumPrimes = new BigInteger[17];  sumPrimes[0] = new BigInteger(primes[0]);  for(int i=1; i<17; ++i){  sumPrimes[i] = new BigInteger(primes[i]);  sumPrimes[i] = sumPrimes[i].multiply(sumPrimes[i-1]);  }    Scanner sc = new Scanner(System.in);  int q = sc.nextInt();  BigInteger n;  for(int i=0; i<q; ++i){  n = sc.nextBigInteger();  for(int j=0; j<17; ++j){  int res = n.compareTo(sumPrimes[j]);  if(res == 0 || res==-1){  System.out.println((res == 0) ? j : j-1);  break;  }  }  }  } |
| Hr Connecting Towns Gandalf is travelling from **Rohan** to **Rivendell** to meet Frodo but there is no direct route from **Rohan** (T1) to **Rivendell** (Tn).  But there are towns T2,T3,T4...Tn-1 such that there are N1 routes from Town T1 to T2, and in general, Ni routes from Tito Ti+1 for i=1 to n-1 and 0 routes for any other Ti to Tj for j ≠ i+1  Find the total number of routes Gandalf can take to reach Rivendell from Rohan.  **Note**  Gandalf has to pass all the towns Ti for i=1 to n-1 in numerical order to reach Tn.  For each Ti , Ti+1 there are only Ni distinct routes Gandalf can take.  **Input Format**  The first line contains an integer T, T test-cases follow.  Each test-case has 2 lines. The first line contains an integer N (the number of towns).  The second line contains N - 1 space separated integers where the ith integer denotes the number of routes, Ni, from the town Ti to Ti+1  **Output Format**  Total number of routes from T1 to Tn modulo 1234567  [http://en.wikipedia.org/wiki/Modular\_arithmetic](https://www.hackerrank.com/external_redirect?to=http://en.wikipedia.org/wiki/Modular_arithmetic)  **Constraints**  1 <= T<=1000 2< N <=100 1 <= Ni <=1000  **Sample Input**  2  3  1 3  4  2 2 2  **Sample Output**  3  8  **Explanation**  Case 1: 1 route from T1 to T2, 3 routes from T2 to T3, hence only 3 routes.  Case 2: There are 2 routes from each city to the next, at each city, Gandalf has 2 choices to make, hence 2 \* 2 \* 2 = 8.  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  for(int i=0; i<T; ++i){  int N = sc.nextInt();  int res = 1;  for(int j=1; j<N; ++j){  res = (res \* sc.nextInt()) % 1234567;  }  System.out.println(res);  }  } |
| Hr Cutting Paper Squares Mary has an n×m piece of paper that she wants to cut into 1×1 pieces according to the following rules:   * She can only cut *one piece of paper at a time*, meaning she *cannot* fold the paper or layer already-cut pieces on top of one another. * Each cut is a straight line from one side of the paper to the other side of the paper. For example, the diagram below depicts the three possible ways to cut a 3×2  piece of paper:  example-cutting-squares.png   Given n and m, find and print the minimum number of cuts Mary must make to cut the paper into n·m squares that are 1×1  unit in size.  **Input Format**  A single line of two space-separated integers denoting the respective values of n and m.  **Constraints**  1 <= n, m <= 109  **Output Format**  Print a long integer denoting the minimum number of cuts needed to cut the entire paper into 1×1  squares.  **Sample Input**  3 1  **Sample Output**  2  **Explanation**  Mary first cuts the 3×1 piece of paper into a 1×1 piece and a 2×1 piece. She then cuts the 2×1  piece into two 1×1 pieces:  cutting-paper-squares.png  Because it took her two cuts to get n×m =3 pieces of size 1×1, we print 2 as our answer.  Java1  static long solve(int n, int m){  return (long)n\*(m-1) + (n-1);  }  public static void main(String[] args) {  Scanner in = new Scanner(System.in);  int n = in.nextInt();  int m = in.nextInt();  long result = solve(n, m);  System.out.println(result);  } |
| Hr Reverse Game Akash and Akhil are playing a game. They have N balls numbered from 0 to N-1. Akhil asks Akash to reverse the position of the balls, i.e., to change the order from say, 0,1,2,3 to 3,2,1,0. He further asks Akash to reverse the position of the balls N times, each time starting from one position further to the right, till he reaches the last ball. So, Akash has to reverse the positions of the ball starting from 0th position, then from 1st position, then from 2nd position and so on. At the end of the game, Akhil will ask Akash the final position of any ball numbered K. Akash will win the game, if he can answer. Help Akash.  **Input Format**  The first line contains an integer T, i.e., the number of the test cases.  The next T lines will contain two integers N and K.  **Output Format**  Print the final index of ball K in the array.  **Constraints**   1 <= T <= 50  1 <= N <= 105  0 <= K < N  **Sample Input**  2  3 1  5 2  **Sample Output**  2  4  **Explanation**  For first test case, The rotation will be like this:  0 1 2 -> 2 1 0 -> 2 0 1 -> 2 0 1 So, Index of 1 will be 2.  Java1  public static void main(String[] args) {  Scanner in = new Scanner(System.in);  int t = in.nextInt();  for(int i=0; i<t; ++i){  int n = in.nextInt();  int k = in.nextInt();  if(k < n/2)  System.out.println(k\*2+1);  else  System.out.println((n-k-1)\*2);  }  } |
| Hr Strange Grid Again A strange grid has been recovered from an old book. It has 5 columns and infinite number of rows. The bottom row is considered as the first row. First few rows of the grid are like this:  ..............  ..............  20 22 24 26 28  11 13 15 17 19  10 12 14 16 18  1 3 5 7 9  0 2 4 6 8  The grid grows upwards forever!  Your task is to find the integer in cth column in rth row of the grid.  **Input Format**  There will be two integers r and c separated by a single space.  **Constraints**   * 1 <= r <= 2\*109 * 1 <= c <= 5   Rows are indexed from bottom to top and columns are indexed from left to right.  **Output Format**  Output the answer in a single line.  **Sample Input**  6 3  **Sample Output**  25  **Explanation**  The number in the 6th row and 3rd column is 25.  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  long r = sc.nextLong();  long c = sc.nextLong();  if(r % 2 == 0)  System.out.println( (((r-1)/2) \* 5 + c) \* 2 - 1 );  else  System.out.println( (((r-1)/2) \* 5 + c - 1) \* 2 );  } |
| Hr Filling Jars Animesh has N empty candy jars, numbered from 1 to N, with infinite capacity. He performs M operations. Each operation is described by 3 integers, a, b, and k. Here, a and b are indices of the jars, and k is the number of candies to be added inside each jar whose index lies between a and b (both inclusive). Can you tell the average number of candies after M operations?  **Input Format**  The first line contains two integers, N and M, separated by a single space.  M lines follow; each of them contains three integers, a, ,b and k, separated by spaces.  **Constraints**   3 <= N <= 107  1 <= M <= 105  1 <=a<=b<= N 0 <= k <= 106  **Output Format**  A single line containing the average number of candies across N jars, *rounded down* to the nearest integer.  **Note**: *Rounded down* means finding the greatest integer which is less than or equal to the given number. E.g. *13.65* and *13.23* are rounded down to *13*, while *12.98* is rounded down to *12*.  **Sample Input**  5 3  1 2 100  2 5 100  3 4 100  **Sample Output**  160  **Explanation**  Initially each of the jars contains *0* candies  0 0 0 0 0  First operation:  100 100 0 0 0  Second operation:  100 200 100 100 100  Third operation:  100 200 200 200 100  Total = *800*, Average = *800/5* = *160*  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  long N = sc.nextLong();  long M = sc.nextLong();  long res = 0;  for(int i=0; i<M; ++i){  long a = sc.nextLong();  long b = sc.nextLong();  long k = sc.nextLong();  res += (b-a+1) \* k;  }  System.out.println(res/N);  } |
| 628. Maximum Product of Three Numbers  Given an integer array, find three numbers whose product is maximum and output the maximum product.  **Example 1:**  **Input:** [1,2,3]  **Output:** 6  **Example 2:**  **Input:** [1,2,3,4]  **Output:** 24  **Note:**   1. The length of the given array will be in range [3,104] and all elements are in the range [-1000, 1000]. 2. Multiplication of any three numbers in the input won't exceed the range of 32-bit signed integer.   Java1  public int maximumProduct(int[] nums) {  Arrays.sort(nums);  int n = nums.length-1;  int m1 = nums[n-2]\*nums[n-1]\*nums[n];  int m2 = nums[0]\*nums[1]\*nums[n];  return Math.max(m1, m2);  } |
|  |

# **数据结构(Data Structure**)

## 线性结构

### 线性表(Linear List)

|  |
| --- |
| 21. 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.  C++1 (19ms 2.60%)  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;  }  }  C++2 (12ms)  ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {  ListNode\* min;  ListNode\* max;  if(l1 && l2)  {  min = l1->val<=l2->val? l1:l2;  max = l2->val>=l1->val? l2:l1;  }  else if(l1)  {  return l1;  }  else  {  return l2;  }  ListNode\* head = min;  ListNode\* temp;  while(min&&max&&min->next)  {  if(min->next->val <= max->val)  min = min->next;  else  {  temp = min->next;  min->next = max;  min = max;  max = temp;  }  }  min->next = max;  return head;  }  Java1 (1ms)  public ListNode mergeTwoLists(ListNode l1, ListNode l2) {  ListNode dummy = new ListNode(0);  ListNode pre = dummy;  while(l1!=null && l2!=null)  {  if(l1.val < l2.val)  {  pre.next=l1;  l1 = l1.next;  }  else  {  pre.next = l2;  l2 = l2.next;  }  pre = pre.next;  }  if(l1!=null)  {  pre.next = l1;  }  else if(l2!=null)  {  pre.next = l2;  }  return dummy.next;  } |
| 141. 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?  C++1 (9ms) O(n)  bool hasCycle(ListNode \*head) {  ListNode\* slow = head;  ListNode\* fast = head;  while(fast!=NULL && fast->next!=NULL)  {  slow = slow->next;  fast = fast->next->next;  if(slow == fast) return true;  }  return false;  }  C++2 (14ms) O(n)  bool hasCycle(ListNode \*head) {  if(!head) return false;  if(!head->next) return false;  ListNode\*p1 = head;  ListNode\*p2 = head->next->next;  while(p2 && p2->next)  {  if(p1==p2)  return true;  p1=p1->next;  p2=p2->next->next;  }  return false;  }  Java1 (1ms) O(n)  public boolean hasCycle(ListNode head) {  ListNode slow=head, fast=head;  while(fast!=null && fast.next!=null)  {  slow=slow.next;  fast=fast.next.next;  if(slow==fast) return true;  }  return false;  } |
| 142. 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?  C++1  ListNode \*detectCycle(ListNode \*head) {  if(head==NULL) return NULL;  ListNode \*slow=head, \*fast=head;  while(fast!=NULL && fast->next!=NULL)  {  slow=slow->next;  fast=fast->next->next;  if(slow==fast) break;  }  if(fast==NULL || fast->next==NULL) return NULL;  slow=head;  while(slow!=fast)  {  slow = slow->next;  fast = fast->next;  }  return slow;  } |
| 257. 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"]  C++1 (9ms) recursive O(n)space, O(n)time  void TraceTreePaths(vector<string>&res, string aPath, TreeNode\* root)  {  if(root->left==NULL && root->right==NULL) res.push\_back(aPath);  else  {  if(root->left!=NULL) TraceTreePaths(res, aPath+"->"+to\_string(root->left->val), root->left);  if(root->right!=NULL) TraceTreePaths(res, aPath+"->"+to\_string(root->right->val), root->right);  }  }  vector<string> binaryTreePaths(TreeNode\* root) {  vector<string> res;  if(root==NULL) return res;  TraceTreePaths(res, to\_string(root->val), root);  return res;  }  C++2 (4ms) stack (log(n)) iteration O(n)space, O(n)time  vector<string> binaryTreePaths(TreeNode\* root) {  vector<string> res;  if (root == NULL) return res;  stack<TreeNode\*> s;  stack<string> pathStack;  s.push(root);  pathStack.push(to\_string(root->val));  while (!s.empty()) {  TreeNode \* curNode = s.top(); s.pop();  string tmpPath = pathStack.top(); pathStack.pop();  if (curNode->left == NULL && curNode->right == NULL) {  res.push\_back(tmpPath); continue;  }  if (curNode->left != NULL) {  s.push(curNode->left);  pathStack.push(tmpPath + "->" + to\_string(curNode->left->val));  }  if (curNode->right != NULL) {  s.push(curNode->right);  pathStack.push(tmpPath + "->" + to\_string(curNode->right->val));  }  }  return res;  }  Java1 (2ms) O(n)space, O(n)time  private List<String> res;  public List<String> binaryTreePaths(TreeNode root) {  res = new ArrayList<String>();  if(root!=null) inputPath(root, "");  return res;  }    public void inputPath(TreeNode root, String arow) {  if(root.left==null && root.right==null)  res.add(arow+root.val);  if(root.left!=null)  inputPath(root.left, arow+root.val+"->");  if(root.right!=null)  inputPath(root.right, arow+root.val+"->");  } |
| 203. 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  C++1 (6ms)  ListNode\* removeElements(ListNode\* head, int val) {  if(head==NULL) return NULL;  if(head->val == val)  {  ListNode\* newHead = head->next;  head->next = NULL;  delete head;  return removeElements(newHead, val);  }  head->next = removeElements(head->next, val);  return head;  }  C++2 (32ms)  ListNode\* removeElements(ListNode\* head, int val) {  while(head && head->val == val)  {  ListNode \* d1 = head;  head = head->next;  delete d1;  }  if(!head) return head;  ListNode \* p = head;  while(p->next)  {  if(p->next->val == val)  {  ListNode \* d = p->next;  p->next = p->next->next;  delete d;  }  else  {  p = p->next;  }  }  return head;  }  Java1 (2ms)  public ListNode removeElements(ListNode head, int val) {  if(head == null)  return head;  head.next = removeElements(head.next, val);  return (head.val==val) ? head.next : head;  }  Java2 (2ms)  public ListNode removeElements(ListNode head, int val) {  while(head!=null && head.val==val)  {  ListNode del = head;  head = head.next;  del = null;  }  if(head == null)  return head;  head.next = removeElements(head.next, val);  return head;  }  Java2 (2ms)  public ListNode removeElements(ListNode head, int val) {  while(head!=null && head.val==val)  {  ListNode del = head;  head = head.next;  del = null;  }  if(head == null)  return head;  ListNode cur = head;  while(cur.next!=null)  {  if(cur.next.val==val)  {  ListNode del1 = cur.next;  cur.next = del1.next;  del1 = null;  }  else  cur = cur.next;  }  return head;  } |
| 160. 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.   C++1 (52ms) O(m+n)  ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {  ListNode\* lA = headA;  ListNode\* lB = headB;  while(lA!=NULL || lB!=NULL)  {  if(lA == lB) return lA;  lA = (lA != NULL) ? lA->next : headB;  lB = (lB != NULL) ? lB->next : headA;  }  return NULL;  }  C++2 (60ms) O(m+n)  ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {  ListNode \*l1 = headA, \*l2 = headB;  while(l1!=l2)  {  l1 = l1? l1->next:headB;  l2 = l2? l2->next:headA;  }  return l1;  }  Java1 (2ms)  public ListNode getIntersectionNode(ListNode headA, ListNode headB) {  ListNode la = headA, lb=headB;  while(la!=lb)  {  if(la==null) la=headB;  else la = la.next;    if(lb==null) lb = headA;  else lb = lb.next;  }  return la;  } |
| 382. 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 equal probability of returning.  solution.getRandom();  /\*\*  \* Definition for singly-linked list.  \* struct ListNode {  \* int val;  \* ListNode \*next;  \* ListNode(int x) : val(x), next(NULL) {}  \* };  \* Your Solution object will be instantiated and called as such:  \* Solution obj = new Solution(head);  \* int param\_1 = obj.getRandom();  \*/  C++1 (49ms)  class Solution {  private:  ListNode\* list;  public:  /\*\* @param head The linked list's head.  Note that the head is guaranteed to be not null, so it contains at least one node. \*/  Solution(ListNode\* head) {  list = head;  }    /\*\* Returns a random node's value. \*/  int getRandom() {  int res = list->val;  ListNode\* cur = list;  for(int i=1; cur!=NULL; ++i)  {  if(rand()%i == 0) res = cur->val;  cur = cur->next;  }  return res;  }  };  C++2  class Solution {  private:  vector<int> values;  int size;  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) {  while(head!=NULL)  {  values.push\_back(head->val);  head = head->next;  }  size = values.size();  srand(time(NULL));  }    /\*\* Returns a random node's value. \*/  int getRandom() {  return values[rand()%size];  }  };  Java1  public class Solution {  private ListNode myHead;  private Random rand;    /\*\* @param head The linked list's head.  Note that the head is guaranteed to be not null, so it contains at least one node. \*/  public Solution(ListNode head) {  myHead = head;  rand = new Random();  }    /\*\* Returns a random node's value. \*/  public int getRandom() {  ListNode node = myHead;  ListNode res = node;  for (int i=1; node != null; ++i)  {  if(rand.nextInt(i) == 0) //if((int)(Math.random()\*i) == 0)  res = node;  node = node.next;  }  return res.val;  }  }  Java2  public class Solution {  private List<Integer> values;  private int size;  private Random random;  /\*\* @param head The linked list's head. Note that the head is guanranteed to be not null, so it contains at least one node. \*/  public Solution(ListNode head) {  values = new ArrayList<Integer>();  while(head!=null)  {  values.add(head.val);  head = head.next;  }  size = values.size();  random = new Random();  }    /\*\* Returns a random node's value. \*/  public int getRandom() {  return values.get(random.nextInt(size));  }  } |
| 328. 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 ...  C++1 (19ms) O(1) space O(n)  ListNode\* oddEvenList(ListNode\* head) {  if(head==NULL) return head;  ListNode\* odd = head;  ListNode\* evenHead = head->next;  ListNode\* even = evenHead;  while(even != NULL && even->next!=NULL)  {  odd->next = even->next;  even->next = odd->next->next;  odd = odd->next;  even = even->next;  }  odd->next = evenHead;  return head;  }  Java1 (1ms)  public ListNode oddEvenList(ListNode head) {  if(head == null) return null;  ListNode odd = head, even = head.next;  ListNode evenHead = even;  while(even!=null && even.next!=null)  {  odd.next = even.next;  even.next = even.next.next;  odd = odd.next;  even = even.next;  }  odd.next = evenHead;  return head;  } |
| 341. 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].  C++1 (19ms)  /\*\*  \* // This is the interface that allows for creating nested lists.  \* // You should not implement it, or speculate about its implementation  \* class NestedInteger {  \* public:  \* // Return true if this NestedInteger holds a single integer, rather than a nested list.  \* bool isInteger() const;  \*  \* // Return the single integer that this NestedInteger holds, if it holds a single integer  \* // The result is undefined if this NestedInteger holds a nested list  \* int getInteger() const;  \*  \* // Return the nested list that this NestedInteger holds, if it holds a nested list  \* // The result is undefined if this NestedInteger holds a single integer  \* const vector<NestedInteger> &getList() const;  \* };  \*/  class NestedIterator {  public:  stack<NestedInteger> myList;  void PushListIntoStack(vector<NestedInteger> &nestedList)  {  for(int i=nestedList.size()-1; i>=0; --i)  myList.push(nestedList[i]);  }  NestedIterator(vector<NestedInteger> &nestedList) {  PushListIntoStack(nestedList);  }  int next() {  NestedInteger top = myList.top();  myList.pop();  return top.getInteger();  }  bool hasNext() {  if(myList.empty()) return false;  NestedInteger top = myList.top();  if(top.isInteger()) return true;  myList.pop();  PushListIntoStack(top.getList());  return hasNext();  }  };  /\*\*  \* Your NestedIterator object will be instantiated and called as such:  \* NestedIterator i(nestedList);  \* while (i.hasNext()) cout << i.next();  \*/  Java1 (10ms)  public class NestedIterator implements Iterator<Integer> {    private Stack<NestedInteger> stk;    private void pushListToStack(List<NestedInteger> nestedList){  for(int i=nestedList.size()-1; i>=0; --i)  stk.push(nestedList.get(i));  }  public NestedIterator(List<NestedInteger> nestedList) {  stk = new Stack<NestedInteger>();  pushListToStack(nestedList);  }  @Override  public Integer next() {  return stk.pop().getInteger();  }  @Override  public boolean hasNext() {  while(!stk.empty())  {  NestedInteger top = stk.peek();  if(top.isInteger())  return true;  stk.pop();  List<NestedInteger> openList = top.getList();  pushListToStack(openList);  }  return false;  }  }  Testcase:   |  |  |  | | --- | --- | --- | | Id | Testcase | Result | | 1 | [[1,1],[6,7],[2,3]] | [1,1,6,7,2,3] | | 2 | [[-1,-1],[[[]]],[2,3]] | [-1,-1,2,3] | |
| 237. 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 -> 4after calling your function.  C++1 (20ms)  void deleteNode(ListNode\* node) {  ListNode\* p = node->next;  \*node = \*p;  delete p;  }  C++2 (19ms)  void deleteNode(ListNode\* node) {  ListNode\* del = node->next;  node->val = del->val;  node->next = del->next;  delete del;  }  Java1 (1ms)  public void deleteNode(ListNode node) {  node.val = node.next.val;  node.next = node.next.next;  } |
| 206. Reverse Linked List  Reverse a singly linked list.  [click to show more hints.](https://leetcode.com/problems/reverse-linked-list/)  **Hint:**  A linked list can be reversed either iteratively or recursively. Could you implement both?  C++1 (8ms)  ListNode\* reverseList(ListNode\* head) {  ListNode\* p1 = NULL;  ListNode \* Next;  while(head)  {  Next = head->next;  head->next = p1;  p1 = head;  head = Next;  }  return p1;  }  C++2 (12ms)  ListNode\* reverseList(ListNode\* head) {  if(!head || !head->next)  return head;  ListNode\* p1 = reverseList(head->next);  head->next->next = head;  head->next = NULL;  return p1;  } |
| 92. 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 ≤ *m* ≤ *n* ≤ length of list.  C++1  ListNode\* reverseBetween(ListNode\* head, int m, int n) {  ListNode dummy(0);  dummy.next = head;  ListNode\* cur=head, \*pre = &dummy, \*nex;  for(int i=1; i<m; ++i)  {  pre = pre->next;  cur = cur->next;  }  for(int i=0; i<n-m; ++i)  {  nex = cur->next;  cur->next = nex->next;  nex->next = pre->next;  pre->next = nex;  }  return dummy.next;  }  C++2  ListNode\* reverseBetween(ListNode\* head, int m, int n) {  ListNode\* newHead = new ListNode(0);  newHead->next = head;  ListNode\* pre = newHead;  for(int i=0; i<m-1; ++i)  {  pre = pre->next;  }  ListNode\* cur = pre->next, \*move;  for(int i=0; i<n-m; ++i)  {  move = cur->next;  cur->next = move->next;  move->next = pre->next;  pre->next = move;  }  return newHead->next;  } |
| 83. 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.  C++1 (9ms)  ListNode\* deleteDuplicates(ListNode\* head) {  if(head == NULL || head->next==NULL) return head;  ListNode\* p = head;  while(p->next != NULL)  {  if(p->val == p->next->val)  {  ListNode \*del = p->next;  p->next = del->next;  delete del;  }  else if(p->next != NULL) p = p->next;  }  return head;  } |
| 82. 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.  C++1  ListNode\* deleteDuplicates(ListNode\* head) {  if(head == NULL || head->next==NULL) return head;  if(head->val == head->next->val)  {  ListNode\* move = head;  while(move->next!=NULL && move->val == move->next->val)  move = move->next;  return deleteDuplicates(move->next);  }  else  head->next = deleteDuplicates(head->next);  return head;  } |
| 2. Add Two Numbers  ou are given two **non-empty** linked lists representing two non-negative integers. 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.  You may assume the two numbers do not contain any leading zero, except the number 0 itself.  **Input:** (2 -> 4 -> 3) + (5 -> 6 -> 4) **Output:** 7 -> 0 -> 8  C++1  ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {  ListNode dummy(0);  ListNode \*pre = &dummy;  int carry=0;  while(l1!=NULL || l2!=NULL || carry>0)  {  int a = l1==NULL ? 0 : l1->val;  int b = l2==NULL ? 0 : l2->val;  int sum = a+b+carry;  carry = sum/10;  ListNode \*node = new ListNode(sum%10);  pre->next = node;  pre = pre->next;  if(l1!=NULL) l1=l1->next;  if(l2!=NULL) l2=l2->next;  }  return dummy.next;  }  C++2  ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {  ListNode head(0), \*pre=&head;  int v1, v2, sum, carry=0;  while(l1!=NULL || l2!=NULL || carry)  {  v1=v2=0;  if(l1!=NULL)  {  v1 = l1->val;  l1=l1->next;  }  if(l2!=NULL)  {  v2 = l2->val;  l2=l2->next;  }  sum = v1+v2+carry;  carry = sum/10;  pre->next = new ListNode(sum%10);  pre = pre->next;  }  return head.next;  }  C++3 using queue |
| 445. Add Two Numbers II  You are given two **non-empty** linked lists representing two non-negative integers. The most significant digit comes first and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.  You may assume the two numbers do not contain any leading zero, except the number 0 itself.  **Follow up:** What if you cannot modify the input lists? In other words, reversing the lists is not allowed.  **Example:**  **Input:** (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)  **Output:** 7 -> 8 -> 0 -> 7  C++1 (49ms) using stack  ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {  stack<int> s1, s2;  while(l1!=NULL)  {  s1.push(l1->val);  l1 = l1->next;  }  while(l2!=NULL)  {  s2.push(l2->val);  l2 = l2->next;  }  ListNode\* head = new ListNode(0);  while(!s1.empty() || !s2.empty())  {  int a=0, b=0;  if(!s1.empty())  {  a = s1.top();  s1.pop();  }  if(!s2.empty())  {  b = s2.top();  s2.pop();  }  int sum = a + b + head->val;  ListNode\* carry = new ListNode(sum/10);  head->val = sum%10;  carry->next = head;  head = carry;  }  return (head->next==NULL || head->val!=0) ? head : head->next;  }  C++2 reverse both of them |
| 24. 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.  C++1  ListNode\* swapPairs(ListNode\* head) {  if(head==NULL || head->next==NULL) return head;  ListNode\* newHead = head->next;  head->next = newHead->next;  newHead->next = head;  head->next = swapPairs(head->next);  return newHead;  }  C++2  ListNode\* swapPairs(ListNode\* head) {  if(!head || !head->next) return head;  ListNode\* p = head->next;  head->next = swapPairs(p->next);  p->next = head;  return p;  }  C++3  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;  }  Java1  public ListNode swapPairs(ListNode head) {  if(head==null || head.next==null) return head;  ListNode dummy = new ListNode(0);  ListNode pre = dummy;  while(head!=null && head.next!=null)  {  pre.next = head.next;  head.next = pre.next.next;  pre.next.next = head;  pre = head;  head = head.next;  }  return dummy.next;  } |
| 116. 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  C++1  void connect(TreeLinkNode \*root) {  if(root==NULL) return;  if(root->left != NULL)  {  root->left->next = root->right;  if(root->next != NULL)  root->right->next = root->next->left;  }  connect(root->left);  connect(root->right);  }  C++2  void connect(TreeLinkNode \*root) {  if(root==NULL) return;  TreeLinkNode \*pre = root;  TreeLinkNode \*cur;  while(pre->left!=NULL)  {  cur = pre;  while(cur!=NULL)  {  cur->left->next = cur->right;  if(cur->next != NULL)  {  cur->right->next = cur->next->left;  }  cur = cur->next;  }  pre = pre->left;  }  } |
| 117. 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  C++1  void connect(TreeLinkNode \*root) {  if(root==NULL) return;  TreeLinkNode dummy(0);  TreeLinkNode \*cur=root, \*pre = &dummy;  while(cur)  {  for(pre=&dummy; cur; cur=cur->next)  {  if(cur->left != NULL)  {  pre->next = cur->left;  pre = pre->next;  }  if(cur->right != NULL)  {  pre->next = cur->right;  pre = pre->next;  }  }  cur = dummy.next; //dummy point to the first left child  dummy.next = NULL;  }  } |
| 19. Remove Nth Node From End of List  Given a linked list, remove the *n*th 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.  C++1  ListNode\* removeNthFromEnd(ListNode\* head, int n) {  ListNode dummy(0);  dummy.next = head;  ListNode\* preDel = &dummy, \*fast=head;  for(int i=0; i<n && fast; ++i)  fast = fast->next;  while(fast!=NULL)  {  preDel = preDel->next;  fast = fast->next;  }  if(preDel->next!=NULL)  {  ListNode\* del = preDel->next;  preDel->next = del->next;  del->next = NULL;  delete del;  }  return dummy.next;  }  C++2  ListNode\* removeNthFromEnd(ListNode\* head, int n) {  ListNode\* p1 = head, \*p2 = head;  for(int i=0; i<n; i++)  p2 = p2->next;  if(p2==NULL)  {  p1=head->next;  head->next = NULL;  return p1;  }  while(p2->next)  {  p1 = p1->next;  p2 = p2->next;  }  p2 = p1->next;  p1->next = p2->next;  p2->next = NULL;  return head;  }  C++3  ListNode\* removeNthFromEnd(ListNode\* head, int n) {  ListNode\*\* p1 = &head, \*p2 = head;  for(int i=0; i<n-1; i++)  p2 = p2->next;  while(p2->next)  {  p1 = &((\*p1)->next);  p2 = p2->next;  }  \*p1 = (\*p1)->next;  return head;  } |
| 147. Insertion Sort List  Sort a linked list using insertion sort.  C++1  ListNode\* insertionSortList(ListNode\* head) {  ListNode dummy(0);  //dummy->next = head; //if has this connection, will create a circle, will be ETL  ListNode\* cur=head, \*pre = &dummy;  while(cur != NULL)  {  ListNode\* temp = cur->next;  if(pre->next==NULL || pre->next->val > cur->val) pre = &dummy;  while(pre->next!=NULL && pre->next->val < cur->val)  pre = pre->next;  cur->next = pre->next;  pre->next = cur;  cur = temp;  }  return dummy.next;  }  C++2  ListNode\* insertionSortList(ListNode\* head) {  if(head==NULL || head->next==NULL) return head;  ListNode\* newHead = new ListNode(0);  ListNode\* cur = head, \*pre = newHead, \*move=NULL;  while(cur!=NULL)  {  move = cur->next;  if(!pre || !pre->next || pre->next->val >= cur->val) pre = newHead;  while(pre->next!=NULL && pre->next->val<cur->val)  pre = pre->next;  cur->next = pre->next;  pre->next = cur;  // pre = newHead;  cur = move;  }  return newHead->next;  }  C++3 (traditional way, very slow)  ListNode\* insertionSortList(ListNode\* head) {  if(head==NULL || head->next==NULL) return head;  ListNode\* newHead = new ListNode(0);  ListNode\* cur = head, \*pre = newHead, \*move=NULL;  while(cur!=NULL)  {  move = cur->next;  while(pre->next!=NULL && pre->next->val<cur->val)  pre = pre->next;  cur->next = pre->next;  pre->next = cur;  pre = newHead;  cur = move;  }  return newHead->next;  } |
| 86. 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.  C++1  ListNode\* partition(ListNode\* head, int x) {  ListNode left(0), right(0);  ListNode \*L=&left, \*R=&right;  while(head != NULL)  {  if(head->val < x)  {  L->next = head;  L = L->next;  }  else  {  R->next = head;  R = R->next;  }  head = head->next;  }  L->next = right.next;  R->next = NULL;  return left.next;  } |
| 148. Sort List  Sort a linked list in *O*(*n* log *n*) time using constant space complexity.  C++1 (TLE) quick sort  class Solution {  public:  ListNode\* merge(ListNode\* left, ListNode\* mid, ListNode\* right)  {  mid->next = right;  if(left==NULL) return mid;    ListNode dummy(0);  dummy.next = left;  while(left->next!=NULL) left = left->next;  left->next = mid;  return dummy.next;  }  ListNode\* sortList(ListNode\* head) {  if(head == NULL || head->next==NULL) return head;  ListNode left(0), right(0);  ListNode \*L=&left, \*R = &right, \*cur=head->next;  while(cur != NULL)  {  if(cur->val <= head->val)  {  L->next = cur;  L = L->next;  }  else  {  R->next = cur;  R = R->next;  }  cur = cur->next;  }  L->next = NULL;  R->next = NULL;  return merge(sortList(left.next), head, sortList(right.next));  }  };  C++2 Divide and conquer  class Solution {  public:  ListNode\* merge(ListNode\* left, ListNode\* right)  {  if(left==NULL) return right;  if(right==NULL) return left;  if(left->val < right->val)  {  left->next = merge(left->next, right);  return left;  }  else  {  right->next = merge(left, right->next);  return right;  }  }  ListNode\* sortList(ListNode\* head) {  if(head==NULL || head->next==NULL) return head;  ListNode \*pre, \*slow=head, \*fast=head;  while(fast!=NULL && fast->next!=NULL)  {  pre = slow;  slow = slow->next;  fast = fast->next->next;  }  pre->next = NULL;  return merge(sortList(head), sortList(slow));  }  }; |
| 143. Reorder List  Given a singly linked list *L*: *L*0→*L*1→…→*Ln*-1→*L*n, reorder it to: *L*0→*Ln*→*L*1→*Ln*-1→*L*2→*Ln*-2→…  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}.  C++1  class Solution {  public:  ListNode\* reverse(ListNode\* head)  {  if(head==NULL|| head->next==NULL) return head;  ListNode\* pre=head, \*cur= head->next;  pre->next=NULL;  while(cur!=NULL)  {  head = cur->next;  cur->next=pre;  pre = cur;  cur = head;  }  return pre;  }  void combine(ListNode\* left, ListNode\* right)  {  if(left==NULL || right==NULL) return;  ListNode\* next=left->next;  left->next=right;  combine(right, next);  }  void reorderList(ListNode\* head) {  if(head==NULL || head->next==NULL) return;  ListNode\* left=head, \*right=head;  while(right!=NULL && right->next!=NULL)  {  left = left->next;  right = right->next->next;  }  right = left->next;  left->next=NULL;  left = head;  right = reverse(right);  combine(left, right);  }  };  C++2  void reorderList(ListNode\* head) {  if(head==NULL || head->next==NULL) return;  ListNode\* p1=head, \*p2=head->next, \*head2;    //find mid  while(p2!=NULL && p2->next!=NULL)  {  p1=p1->next;  p2=p2->next->next;  }    //cut half  head2 = p1->next;  p1->next = NULL;    //reverse right half  p2 = head2->next;  head2->next = NULL;  while(p2!=NULL)  {  p1 = p2->next;  p2->next = head2;  head2 = p2;  p2 = p1;  }    //merge left half and right half  for(p1=head, p2 = head2; p1;)  {  head2 = p1->next;  p1->next = p2;  p1 = p1->next;  p2 = head2;  }  } |
| 138. 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.  C++1  RandomListNode \*copyRandomList(RandomListNode \*head) {  RandomListNode \*newHead, \*p1, \*p2;  if(head==nullptr) return nullptr;  //create list p2, and link p2 in p1 //1->1->2->2->3->3...  for(p1=head; p1!=nullptr; p1=p1->next->next)  {  p2 = new RandomListNode(p1->label);  p2->next = p1->next;  p1->next = p2;  }  //copy random from p1 to p2  for(p1=head; p1!=nullptr; p1 = p1->next->next)  {  p2 = p1->next;  if(p1->random!=nullptr) p2->random = p1->random->next;  }  //cup p1 and p2  newHead = head->next;  for(p1=head; p1!=nullptr; p1=p1->next)  {  p2 = p1->next;  p1->next = p2->next;  if(p1->next!=nullptr) p2->next = p1->next->next;  }  return newHead;  } |
| 61. 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.  C++1  ListNode\* rotateRight(ListNode\* head, int k) {  if(head==NULL) return head;  int len = 0;  ListNode\* cur = head;  while(cur!=NULL)  {  len++;  cur=cur->next;  }  k = k%len;  if(k==0) return head; //corner cases: len=1, k%len==0  ListNode\* slow=head, \*fast=head;  for(int i=0; i<k; ++i)  fast = fast->next;  while(fast->next!=NULL)  {  slow = slow->next;  fast = fast->next;  }  ListNode\* newHead = slow->next;  slow->next = NULL;  fast->next=head;  return newHead;  }  C++2  ListNode\* rotateRight(ListNode\* head, int k) {  if(head==NULL || k==0) return head;  ListNode \*first = head, \*tail = head;  int len = 1;  while(first->next!=NULL)  {  len++;  first = first->next;  }  k %= len;  if(k==0) return head;  for(int i=1; i<len-k; ++i) tail = tail->next;    first->next = head;  head = tail->next;  tail->next = NULL;    return head;  } |
|  |

### 栈(Stack)

|  |
| --- |
| 232. 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).   /\*\*  \* Your MyQueue object will be instantiated and called as such:  \* MyQueue obj = new MyQueue();  \* obj.push(x);  \* int param\_2 = obj.pop();  \* int param\_3 = obj.peek();  \* bool param\_4 = obj.empty();  \*/  C++1 (3ms)  class MyQueue {  private:  stack<int> inputStack;  stack<int> outputStack;  public:  /\*\* Initialize your data structure here. \*/  MyQueue() {    }    /\*\* Push element x to the back of queue. \*/  void push(int x) {  inputStack.push(x);  }    /\*\* Removes the element from in front of queue and returns that element. \*/  int pop() {  int res = peek();  outputStack.pop();  return res;  }    /\*\* Get the front element. \*/  int peek() {  if(outputStack.empty())  {  while(!inputStack.empty())  {  outputStack.push(inputStack.top());  inputStack.pop();  }  }  return outputStack.top();  }    /\*\* Returns whether the queue is empty. \*/  bool empty() {  return outputStack.empty() && inputStack.empty();  }  };  C++2 (0ms)  class Queue {  public:  stack<int> s;  stack<int> sub;    void aStackb(stack<int>& a, stack<int>& b)  {  while(!a.empty())  {  b.push(a.top());  a.pop();  }  }    // Push element x to the back of queue.  void push(int x) {  aStackb(s, sub);  s.push(x);  aStackb(sub, s);  }  // Removes the element from in front of queue.  void pop(void) {  s.pop();  }  // Get the front element.  int peek(void) {  return s.top();  }  // Return whether the queue is empty.  bool empty(void) {  return s.empty();  }  };  Java1 (120ms)  class MyQueue {  private Stack<Integer> input = new Stack();  private Stack<Integer> output = new Stack();  // Push element x to the back of queue.  public void push(int x) {  input.push(x);  }  // Removes the element from in front of queue.  public void pop() {  peek();  output.pop();  }  // Get the front element.  public int peek() {  if(output.empty())  {  while(!input.empty())  {  output.push(input.pop());  }  }  return output.peek();  }  // Return whether the queue is empty.  public boolean empty() {  return input.empty() && output.empty();  }  }  Java2 (117ms)  class MyQueue {  private Stack<Integer> input = new Stack();  private Stack<Integer> output = new Stack();    void aStackb(Stack<Integer> a, Stack<Integer> b)  {  while(!a.empty())  {  b.push(a.pop());  }  }    // Push element x to the back of queue.  public void push(int x) {  aStackb(output, input);  output.push(x);  aStackb(input, output);  }  // Removes the element from in front of queue.  public void pop() {  output.pop();  }  // Get the front element.  public int peek() {  return output.peek();  }  // Return whether the queue is empty.  public boolean empty() {  return output.empty();  }  } |
| 155. Min Stack  Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.   * push(x) -- Push element x onto stack. * pop() -- Removes the element on top of the stack. * top() -- Get the top element. * getMin() -- Retrieve the minimum element in the stack.   **Example:**  MinStack minStack = new MinStack();  minStack.push(-2);  minStack.push(0);  minStack.push(-3);  minStack.getMin(); --> Returns -3.  minStack.pop();  minStack.top(); --> Returns 0.  minStack.getMin(); --> Returns -2.  C++1 (32ms)  class MinStack {  private:  stack<int> thisStack;  stack<int> minStack;  public:  /\*\* initialize your data structure here. \*/  MinStack() {    }    void push(int x) {  if(thisStack.empty())  {  thisStack.push(x);  minStack.push(x);  }  else  {  thisStack.push(x);  if(x<minStack.top())  minStack.push(x);  else  minStack.push(minStack.top());  }  }    void pop() {  thisStack.pop();  minStack.pop();  }    int top() {  return thisStack.top();  }    int getMin() {  return minStack.top();  }  };  /\*\*  \* Your MinStack object will be instantiated and called as such:  \* MinStack obj = new MinStack();  \* obj.push(x);  \* obj.pop();  \* int param\_3 = obj.top();  \* int param\_4 = obj.getMin();  \*/  C++2 (56ms)  class MinStack {  private:  stack<int> myStack;  stack<int> myMin;  public:  /\*\* initialize your data structure here. \*/  MinStack() {  while(!myStack.empty()) myStack.pop();  while(!myMin.empty()) myMin.pop();  }    void push(int x) {  myStack.push(x);  if(myMin.empty() || x<=myMin.top())  {  myMin.push(x);  }  }    void pop() {  if(myStack.empty() || myMin.empty()) return;  if(myStack.top()==myMin.top())  {  myStack.pop();  myMin.pop();  }  else  {  myStack.pop();  }  }    int top() {  if(myStack.empty()) return -1;  return myStack.top();  }    int getMin() {  if(myMin.empty()) return -1;  return myMin.top();  }  };  Java1 (144ms)  public class MinStack {  private Stack<Integer> stack;  private Stack<Integer> minStack;  /\*\* initialize your data structure here. \*/  public MinStack() {  stack = new Stack<Integer>();  minStack = new Stack<>();  }    public void push(int x) {  stack.push(x);  if(minStack.empty())  minStack.push(x);  else  minStack.push(Math.min(minStack.peek(), x));  }    public void pop() {  stack.pop();  minStack.pop();  }    public int top() {  return stack.peek();  }    public int getMin() {  return minStack.peek();  }  } |
| 394. 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".  C++1 (0ms) O(n)  string decodeString(string s) {  int n = 0;  string word="";  stack<int> repeat;  stack<string> words;  for(auto c : s)  {  if(isdigit(c))  n = 10\*n + (c-'0');  else if(c=='[')  {  repeat.push(n);  n=0;  words.push(word);  word.clear();  }  else if(c==']')  {  int rep = repeat.top();  repeat.pop();  while(rep-- > 0)  words.top() += word;  word = words.top();  words.pop();  }  else  word += c;  }  return words.empty() ? word : words.top();  } |
| Codility [Fish](https://codility.com/demo/results/training2XDFR3-Y87/)  You are given two non-empty zero-indexed arrays A and B consisting of N integers. Arrays A and B represent N voracious fish in a river, ordered downstream along the flow of the river.  The fish are numbered from 0 to N − 1. If P and Q are two fish and P < Q, then fish P is initially upstream of fish Q. Initially, each fish has a unique position.  Fish number P is represented by A[P] and B[P]. Array A contains the sizes of the fish. All its elements are unique. Array B contains the directions of the fish. It contains only 0s and/or 1s, where:   * 0 represents a fish flowing upstream, * 1 represents a fish flowing downstream.   If two fish move in opposite directions and there are no other (living) fish between them, they will eventually meet each other. Then only one fish can stay alive − the larger fish eats the smaller one. More precisely, we say that two fish P and Q meet each other when P < Q, B[P] = 1 and B[Q] = 0, and there are no living fish between them. After they meet:   * If A[P] > A[Q] then P eats Q, and P will still be flowing downstream, * If A[Q] > A[P] then Q eats P, and Q will still be flowing upstream.   We assume that all the fish are flowing at the same speed. That is, fish moving in the same direction never meet. The goal is to calculate the number of fish that will stay alive.  For example, consider arrays A and B such that:  A[0] = 4 B[0] = 0 A[1] = 3 B[1] = 1 A[2] = 2 B[2] = 0 A[3] = 1 B[3] = 0 A[4] = 5 B[4] = 0  Initially all the fish are alive and all except fish number 1 are moving upstream. Fish number 1 meets fish number 2 and eats it, then it meets fish number 3 and eats it too. Finally, it meets fish number 4 and is eaten by it. The remaining two fish, number 0 and 4, never meet and therefore stay alive.  Write a function:  int solution(vector<int> &A, vector<int> &B);  that, given two non-empty zero-indexed arrays A and B consisting of N integers, returns the number of fish that will stay alive.  For example, given the arrays shown above, the function should return 2, as explained above.  Assume that:   * N is an integer within the range [1..100,000]; * each element of array A is an integer within the range [0..1,000,000,000]; * each element of array B is an integer that can have one of the following values: 0, 1; * the elements of A are all distinct.   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  Java  **public** **int** **solution**(**int**[] A, **int**[] B) {  // write your code in Java SE 8    Deque<Integer> stack = **new** LinkedList<>();  Deque<Integer> direction = **new** LinkedList<>();  **for**(**int** i=0; i<A.length; ++i){  **while**(!direction.isEmpty() && B[i]==0 && B[i] != direction.peek()){  **if**(A[i]<stack.peek()) **break**;  stack.pop();  direction.pop();  }  **if**(direction.isEmpty() || B[i] == direction.peek() || B[i]==1){  stack.push(A[i]);  direction.push(B[i]);  }  }  **return** stack.size();  }  C++1  **int** **solution**(**vector**<**int**> &A, **vector**<**int**> &B) {  // write your code in C++14 (g++ 6.2.0)  **stack**<**int**> upFish;  **int** downFish=0, n = A.size();  **for**(**int** i=0; i<n; ++i)  {  **if**(B[i] == 1) upFish.push(A[i]); //should be downFish  **else**  {  **while**(!upFish.empty() && upFish.top()<A[i])  upFish.pop();  **if**(upFish.empty()) downFish++;  }  }  **return** downFish + upFish.size();  }  Testcase  extreme\_small  1 or 2 fishes  [▶](https://codility.com/demo/results/training2XDFR3-Y87/)  simple1  simple test  [▶](https://codility.com/demo/results/training2XDFR3-Y87/)  simple2  simple test  [▶](https://codility.com/demo/results/training2XDFR3-Y87/)  small\_random  small random test, N = ~100  [▶](https://codility.com/demo/results/training2XDFR3-Y87/)  medium\_random  small medium test, N = ~5,000  [▶](https://codility.com/demo/results/training2XDFR3-Y87/)  large\_random  large random test, N = ~100,000  [▶](https://codility.com/demo/results/training2XDFR3-Y87/)  extreme\_range1  all except one fish flowing in the same direction  [▶](https://codility.com/demo/results/training2XDFR3-Y87/)  extreme\_range2  all fish flowing in the same direction |
| Codility [Nesting](https://codility.com/demo/results/trainingWGJUTH-UYZ/)  A string S consisting of N characters is called *properly nested* if:   * S is empty; * S has the form "(U)" where U is a properly nested string; * S has the form "VW" where V and W are properly nested strings.   For example, string "(()(())())" is properly nested but string "())" isn't.  Write a function:  int solution(string &S);  that, given a string S consisting of N characters, returns 1 if string S is properly nested and 0 otherwise.  For example, given S = "(()(())())", the function should return 1 and given S = "())", the function should return 0, as explained above.  Assume that:   * N is an integer within the range [0..1,000,000]; * string S consists only of the characters "(" and/or ")".   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(1) (not counting the storage required for input arguments).   Java  **public** **int** **solution**(String S) {  // write your code in Java SE 8  Deque<Character> stack = **new** LinkedList<>();  **for**(**char** c : S.toCharArray()){  **if**(c == ')'){  **if**(stack.isEmpty()) **return** 0;  **if**(!stack.isEmpty() && stack.pop() != '(') **return** 0;  }  **else** stack.push(c);  }  **return** stack.isEmpty() ? 1 : 0;  }  C++1  **int** **solution**(**string** &S) {  // write your code in C++14 (g++ 6.2.0)  **int** left = 0;  **for**(**char** c : S)  {  **if**(c=='(') left++;  **else** **if**(--left < 0) return 0;  }  **return** left==0 ? 1 : 0;  }  Testcase  negative\_match  invalid structure, but the number of parentheses matches  [▶](https://codility.com/demo/results/trainingWGJUTH-UYZ/)  empty  empty string  [▶](https://codility.com/demo/results/trainingWGJUTH-UYZ/)  simple\_grouped  simple grouped positive and negative test, length=22  [▶](https://codility.com/demo/results/trainingWGJUTH-UYZ/)  small\_random  [▶](https://codility.com/demo/results/trainingWGJUTH-UYZ/)  large1  simple large positive and negative test, 10K or 10K+1 ('s followed by 10K )'s  [▶](https://codility.com/demo/results/trainingWGJUTH-UYZ/)  large\_full\_ternary\_tree  tree of the form T=(TTT) and depth 11, length=177K+  [▶](https://codility.com/demo/results/trainingWGJUTH-UYZ/)  multiple\_full\_binary\_trees  sequence of full trees of the form T=(TT), depths [1..10..1], with/without unmatched ')' at the end, length=49K+  [▶](https://codility.com/demo/results/trainingWGJUTH-UYZ/)  broad\_tree\_with\_deep\_paths  string of the form (TTT...T) of 300 T's, each T being '(((...)))' nested 200-fold, length=1 million |
| Codility [StoneWall](https://codility.com/demo/results/trainingMY6QRE-CSN/)  You are going to build a stone wall. The wall should be straight and N meters long, and its thickness should be constant; however, it should have different heights in different places. The height of the wall is specified by a zero-indexed array H of N positive integers. H[I] is the height of the wall from I to I+1 meters to the right of its left end. In particular, H[0] is the height of the wall's left end and H[N−1] is the height of the wall's right end.  The wall should be built of cuboid stone blocks (that is, all sides of such blocks are rectangular). Your task is to compute the minimum number of blocks needed to build the wall.  Write a function:  int solution(vector<int> &H);  that, given a zero-indexed array H of N positive integers specifying the height of the wall, returns the minimum number of blocks needed to build it.  For example, given array H containing N = 9 integers:  H[0] = 8 H[1] = 8 H[2] = 5 H[3] = 7 H[4] = 9 H[5] = 8 H[6] = 7 H[7] = 4 H[8] = 8  the function should return 7. The figure shows one possible arrangement of seven blocks.  https://codility-frontend-prod.s3.amazonaws.com/media/task_static/stone_wall/static/images/auto/4f1cef49cc46d451e88109d449ab7975.png  Assume that:   * N is an integer within the range [1..100,000]; * each element of array H is an integer within the range [1..1,000,000,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &H) {  // write your code in C++14 (g++ 6.2.0)  **int** res = 0;  **stack**<**int**> sta;  **for**(**int** h : H)  {  **while**(!sta.empty() && sta.top()>h)  {  sta.pop();  res++;  }  **if**(sta.empty() || sta.top() < h) sta.push(h);  }  **return** res + sta.size();  }  Testcase  large\_piramid  [▶](https://codility.com/demo/results/trainingMY6QRE-CSN/)  large\_increasing\_decreasing  [▶](https://codility.com/demo/results/trainingMY6QRE-CSN/)  large\_up\_to\_20  [▶](https://codility.com/demo/results/trainingMY6QRE-CSN/)  large\_up\_to\_100  [▶](https://codility.com/demo/results/trainingMY6QRE-CSN/)  large\_max |
| 385. 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.  C++1  /\*\*  \* // This is the interface that allows for creating nested lists.  \* // You should not implement it, or speculate about its implementation  \* class NestedInteger {  \* public:  \* // Constructor initializes an empty nested list.  \* NestedInteger();  \*  \* // Constructor initializes a single integer.  \* NestedInteger(int value);  \*  \* // Return true if this NestedInteger holds a single integer, rather than a nested list.  \* bool isInteger() const;  \*  \* // Return the single integer that this NestedInteger holds, if it holds a single integer  \* // The result is undefined if this NestedInteger holds a nested list  \* int getInteger() const;  \*  \* // Set this NestedInteger to hold a single integer.  \* void setInteger(int value);  \*  \* // Set this NestedInteger to hold a nested list and adds a nested integer to it.  \* void add(const NestedInteger &ni);  \*  \* // Return the nested list that this NestedInteger holds, if it holds a nested list  \* // The result is undefined if this NestedInteger holds a single integer  \* const vector<NestedInteger> &getList() const;  \* };  \*/  class Solution {  public:  NestedInteger deserialize(string s) {  if(s[0] != '[') return NestedInteger(stoi(s));  stack<NestedInteger\*> sta;  for(int left=0, i=0; i<s.length(); ++i)  {  char c = s[i];  if(c=='[')  {  sta.push(new NestedInteger());  left = i+1;  }  else if(c==',' || c==']')  {  if(left != i)  {  int num = stoi(s.substr(left, i-left));  sta.top()->add(NestedInteger(num));  }  if(c==']' && sta.size() > 1)  {  NestedInteger\* temp = sta.top();  sta.pop();  sta.top()->add(\*temp);  }  left = i+1;  }  }  return \*sta.top();  }  }; |
| 456. 132 Pattern  Given a sequence of n integers a1, a2, ..., an, a 132 pattern is a subsequence a**i**, a**j**, a**k** such that **i** < **j** < **k** and a**i** < a**k** < a**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] and [-1, 2, 0].  C++1  bool find132pattern(vector<int>& nums) {  int s3 = INT\_MIN;  stack<int> minHeap;  for(int i=nums.size()-1; i>=0; --i)  {  if(nums[i] < s3) return true;  else  {  while(!minHeap.empty() && nums[i] > minHeap.top())  {  s3 = minHeap.top(); //the largest s3 which is less than nums[i]  minHeap.pop();  }  minHeap.push(nums[i]);  }  }  return false;  } |
| 150. Evaluate Reverse Polish Notation  Evaluate the value of an arithmetic expression in [Reverse Polish Notation](http://en.wikipedia.org/wiki/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  C++1  int evalRPN(vector<string>& tokens) {  stack<int> sta;  for(string token : tokens)  {  if(token == "+" || token == "-" || token == "\*" || token == "/")  {  int b = sta.top();  sta.pop();  int a = sta.top();  sta.pop();  if(token == "/") sta.push(a/b);  else if(token == "\*") sta.push(a\*b);  else if(token == "-") sta.push(a-b);  else sta.push(a+b);  }  else  sta.push(stoi(token));  }  return sta.top();  } |
| 402. 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 10002 and will be ≥ *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 which is 0.  C++1  string removeKdigits(string num, int k) {  int size=num.length();  int digitS = size-k;  char sta[size];  int i=0, top=-1;  for(; i<size; ++i)  {  while(top>-1 && sta[top]>num[i] && k>0)  {  top--;  k--;  }  sta[++top] = num[i];  }  int start=0;  while(start<digitS && sta[start]=='0') //check start position first  start++;  return start==digitS ? "0" : string(sta, start, digitS-start);  } |
| 682. Baseball Game  You're now a baseball game point recorder.  Given a list of strings, each string can be one of the 4 following types:   1. Integer (one round's score): Directly represents the number of points you get in this round. 2. "+" (one round's score): Represents that the points you get in this round are the sum of the last two valid round's points. 3. "D" (one round's score): Represents that the points you get in this round are the doubled data of the last valid round's points. 4. "C" (an operation, which isn't a round's score): Represents the last valid round's points you get were invalid and should be removed.   Each round's operation is permanent and could have an impact on the round before and the round after.  You need to return the sum of the points you could get in all the rounds.  **Example 1:**  **Input:** ["5","2","C","D","+"]  **Output:** 30  **Explanation:**  Round 1: You could get 5 points. The sum is: 5.  Round 2: You could get 2 points. The sum is: 7.  Operation 1: The round 2's data was invalid. The sum is: 5.  Round 3: You could get 10 points (the round 2's data has been removed). The sum is: 15.  Round 4: You could get 5 + 10 = 15 points. The sum is: 30.  **Example 2:**  **Input:** ["5","-2","4","C","D","9","+","+"]  **Output:** 27  **Explanation:**  Round 1: You could get 5 points. The sum is: 5.  Round 2: You could get -2 points. The sum is: 3.  Round 3: You could get 4 points. The sum is: 7.  Operation 1: The round 3's data is invalid. The sum is: 3.  Round 4: You could get -4 points (the round 3's data has been removed). The sum is: -1.  Round 5: You could get 9 points. The sum is: 8.  Round 6: You could get -4 + 9 = 5 points. The sum is 13.  Round 7: You could get 9 + 5 = 14 points. The sum is 27.  **Note:**   The size of the input list will be between 1 and 1000.   Every integer represented in the list will be between -30000 and 30000.  C++1  class Solution {  public:  bool isNum(string num)  {  return isdigit(num[0]) || num[0]=='-';  }    int calPoints(vector<string>& ops) {  int res=0, cur=0;  vector<int> nums;  for(string n : ops)  {  int s = nums.size();  if(s==0 && !isNum(n)) continue;  if(n == "C")  {  res -= nums.back();  nums.pop\_back();  }  else if(n == "+" && s > 1)  {  cur = nums[s-2] + nums[s-1];  res += cur;  nums.push\_back(cur);  }  else if(n == "D")  {  cur = 2 \* nums[s-1];  res += cur;  nums.push\_back(cur);  }  else if(isNum(n))  {  cur = stoi(n);  res += cur;  nums.push\_back(cur);  }  }  return res;  }  }; |
|  |

### 队列(Queue)

#### Queue

|  |
| --- |
| 637. Average of Levels in Binary Tree  Given a non-empty binary tree, return the average value of the nodes on each level in the form of an array.  **Example 1:**  **Input:**  3  / \  9 20  / \  15 7  **Output:** [3, 14.5, 11]  **Explanation:**  The average value of nodes on level 0 is 3, on level 1 is 14.5, and on level 2 is 11. Hence return [3, 14.5, 11].  **Note:**   1. The range of node's value is in the range of 32-bit signed integer.   C++1  vector<double> averageOfLevels(TreeNode\* root) {  vector<double> res;  if(root==NULL) return res;  queue<TreeNode\*> q;  q.push(root);  while(!q.empty())  {  int size = q.size();  double ave = 0;  for(int i=0; i<size; ++i)  {  TreeNode\* node = q.front();  q.pop();  ave += node->val;  if(node->left != NULL) q.push(node->left);  if(node->right != NULL) q.push(node->right);  }  res.push\_back(ave/size);  }  return res;  }  Java1  public List<Double> averageOfLevels(TreeNode root) {  List<Double> res = new LinkedList<>();  if(root == null) return res;  Queue<TreeNode> q = new LinkedList<>();  q.add(root);  while(!q.isEmpty()){  int size = q.size();  double ave = 0.0;  for(int i=0; i<size; ++i){  TreeNode node = q.remove();  ave += node.val;  if(node.left != null) q.add(node.left);  if(node.right != null) q.add(node.right);  }  res.add(ave/size);  }  return res;  } |
|  |
|  |

#### 优先队列(Priority Queue)

|  |
| --- |
| 347. 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 ≤ *k* ≤ number of unique elements. * Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size.   C++1 (19ms)  vector<int> topKFrequent(vector<int>& nums, int k) {  unordered\_map<int, int> count;  for(auto num : nums)  count[num]++;  priority\_queue<int, vector<int>, greater<int>> pq;  for(auto ele : count)  {  pq.push(ele.second);  if(pq.size()>k) pq.pop();  }  vector<int>res;  for(auto ele : count)  if(ele.second >= pq.top())  res.push\_back(ele.first);  return res;  }  C++2  vector<int> topKFrequent(vector<int>& nums, int k) {  unordered\_map<int, int> counts;  for(int n : nums) ++counts[n];  vector<vector<int>> buckets(nums.size()+1); // no 0 frequent elements  for(auto m : counts)  {  buckets[m.second].push\_back(m.first);  }  vector<int> res;  for(int i=buckets.size()-1; i>=0; --i)  {  for(int j=0; j<buckets[i].size(); ++j)  {  res.push\_back(buckets[i][j]);  if(res.size()>=k) return res;  }  }  return res;  }  Java1  public List<Integer> topKFrequent(int[] nums, int k) {  Map<Integer, Integer> freqMap = new HashMap<>();  PriorityQueue<Map.Entry<Integer, Integer>> pq =  new PriorityQueue<>((a, b) -> a.getValue() - b.getValue());  List<Integer> res = new ArrayList<>();    for(int num : nums)  {  int count = freqMap.getOrDefault(num, 0);  freqMap.put(num, count + 1);  }    for(Map.Entry<Integer, Integer> map : freqMap.entrySet())  {  pq.offer(map);  if(pq.size()>k)  pq.poll();  }    while(!pq.isEmpty())  {  int num = pq.poll().getKey();  res.add(num);  }  return res;  }  Java2  public List<Integer> topKFrequent(int[] nums, int k) {  int len = nums.length;  Map<Integer, Integer> freqMap = new HashMap<>();  List<Integer> [] buckets = new List[len+1];  List<Integer> res = new ArrayList<>();    for(int num : nums)  {  freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);  }    for(int key : freqMap.keySet())  {  int freq = freqMap.get(key);  if(buckets[freq] == null)  {  buckets[freq] = new ArrayList();  }  buckets[freq].add(key);  }  /\*  for(Map.Entry<Integer, Integer> map : freqMap.entrySet())  {  int freq = map.getValue();  if(buckets[freq] == null)  {  buckets[freq] = new ArrayList();  }  buckets[freq].add(map.getKey());  }  \*/    for(int i=len; i>0; --i)  {  if(buckets[i]==null) continue;  for(int j=0; j<buckets[i].size(); ++j)  {  res.add(buckets[i].get(j));  if(res.size()>= k)  return res;  }  }  return res;  } |
| 407. 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.  https://leetcode.com/static/images/problemset/rainwater_empty.png 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.  https://leetcode.com/static/images/problemset/rainwater_fill.png After the rain, water are trapped between the blocks. The total volume of water trapped is 4.  C++1  class Solution {  public:  struct Cell{  int row;  int col;  int height;  Cell(int r, int c, int h) : row(r), col(c), height(h) {}  };  int trapRainWater(vector<vector<int>>& heightMap) {  auto myComp = []( Cell a, Cell b ) { return a.height > b.height; };  priority\_queue<Cell, vector<Cell>, decltype(myComp)> pq(myComp);  int h = heightMap.size();  int w = (h>0) ? heightMap[0].size() : 0;  vector<vector<bool>> visited(h, vector<bool>(w));  for(int j=0; j<w; ++j)  {  visited[0][j] = visited[h-1][j] = true;  pq.push(Cell(0, j, heightMap[0][j]));  pq.push(Cell(h-1, j, heightMap[h-1][j]));  }  for(int i=0; i<h; ++i)  {  visited[i][0] = visited[i][w-1] = true;  pq.push(Cell(i, 0, heightMap[i][0]));  pq.push(Cell(i, w-1, heightMap[i][w-1]));  }    int res = 0;  int edges[4][2] = {{-1,0},{1,0},{0,-1}, {0,1}};  while(!pq.empty())  {  Cell cell = pq.top();  pq.pop();  for(int i=0; i<4; ++i)  {  int row = cell.row + edges[i][0];  int col = cell.col + edges[i][1];  if(row>=0 && row<h && col>=0 && col<w && !visited[row][col])  {  visited[row][col] = true;  res += max(0, cell.height-heightMap[row][col]);  pq.push(Cell(row, col, max(cell.height, heightMap[row][col])));  }  }  }  return res;  }  }; |
| 373. 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 **(u1,v1),(u2,v2) ...(uk,vk)** 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]  C++1  class Solution {  public:  struct myOper  {  bool operator() (pair<int,int>&a, pair<int,int>&b)  {  return a.first+a.second < b.first+b.second;  }  };  vector<pair<int, int>> kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k) {  vector<pair<int, int>> res;  int n1=nums1.size(), n2=nums2.size();  if(n1==0 || n2==0 || k==0) return res;  priority\_queue<pair<int,int>, vector<pair<int,int>>, myOper> pq;  for(int i=0; i<min(n1,k); ++i)  {  for(int j=0; j<min(n2,k); ++j)  {  if(pq.size()<k)  {  pq.push(make\_pair(nums1[i], nums2[j]));  }  else if(nums1[i]+nums2[j]<pq.top().first + pq.top().second)  {  pq.pop();  pq.push(make\_pair(nums1[i], nums2[j]));  }  }  }  while(!pq.empty())  {  res.push\_back(pq.top());  pq.pop();  }  return res;  }  };  C++2  vector<pair<int, int>> kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k) {  int n1=nums1.size(), n2=nums2.size();  vector<pair<int, int>> res;  if(n1==0 || n2==0) return res;  auto mycomp = [&nums1, &nums2](pair<int, int> a, pair<int, int> b)  { return nums1[a.first]+nums2[a.second] > nums1[b.first]+nums2[b.second]; };  priority\_queue<pair<int, int>, vector<pair<int, int>>, decltype(mycomp)> min\_heap(mycomp);  min\_heap.emplace(0,0);  //min\_heap.push(make\_pair(0,0));  while(res.size()<k && !min\_heap.empty())  {  pair<int, int> minp = min\_heap.top();  min\_heap.pop();  //res.push\_back(make\_pair(nums1[minp.first], nums2[minp.second]));  res.emplace\_back(nums1[minp.first], nums2[minp.second]);  if(minp.first+1<n1)  min\_heap.emplace(minp.first+1, minp.second);  if(minp.first==0 && minp.second+1<n2)  min\_heap.emplace(0, minp.second+1);  }  return res;  } |
|  |

### 数组(Array)

|  |
| --- |
| 27. Remove Element  Given an array and a value, remove all instances of that value in place and return the new length.  Do not allocate extra space for another array, you must do this in place with constant memory.  The order of elements can be changed. It doesn't matter what you leave beyond the new length.  **Example:** Given input array *nums* = [3,2,2,3], *val* = 3  Your function should return length = 2, with the first two elements of *nums* being 2.  C++1 ()  int removeElement(vector<int>& nums, int val) {  int i=0;  for(int j=0; j<nums.size(); ++j)  {  if(nums[j] != val)  nums[i++] = nums[j];  }  return i;  }  C++2  int removeElement(vector<int>& nums, int val) {  nums.erase(remove(nums.begin(), nums.end(), val),nums.end());  return nums.size();  }  Java1  public int removeElement(int[] nums, int val) {  int i=0;  for(int j=0; j<nums.length; ++j)  {  if(nums[j]!=val)  {  if(i!=j) nums[i++]=nums[j];  else i++;  }  }  return i;  }  Java2  **public int removeElement(int[] nums, int val) {**  **int len = A.length;**  **for (int i = 0 ; i< len; ++i){**  **while (A[i]==elem && i< len) {**  **A[i]=A[--len];**  **}**  **}**  **return len;**  **}** |
| 66. Plus One  Given a non-negative integer represented as a **non-empty** array of digits, plus one to the integer.  You may assume the integer do not contain any leading zero, except the number 0 itself.  The digits are stored such that the most significant digit is at the head of the list.  C++1 (3ms)  vector<int> plusOne(vector<int>& digits) {  for(int i=digits.size()-1; i>=0;--i)  {  if(digits[i] != 9)  {  digits[i]++;  return digits;  }  else digits[i]=0;  }  digits.insert(digits.begin(), 1);  return digits;  }  C++2 (4ms)  vector<int> plusOne(vector<int>& digits) {  int i = digits.size();  if(!i)  {  digits.push\_back(1);  return digits;  }  int res = digits[i-1] +1;  int carry = res/10;  digits[--i] = res%10;  while(carry && i>0)  {  res = digits[--i] +carry;  carry = res/10;  digits[i] = res%10;  }  if(carry)  digits.insert(digits.begin(),carry);  return digits;  }  Java1 (0ms)  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[] newDigits = new int[n+1];  newDigits[0]=1;  return newDigits;  } |
| 118. 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]  ]  C++1 (3ms)  vector<vector<int>> generate(int numRows) {  vector<vector<int>> res(numRows, vector<int>(1, 1));  for(int i=1; i<numRows; ++i)  {  for(int j=0; j<res[i-1].size(); ++j)  {  if(j==res[i-1].size()-1) res[i].push\_back(res[i-1][j]);  else res[i].push\_back(res[i-1][j] + res[i-1][j+1]);  }  }  return res;  }  C++2 (0ms)  vector<vector<int>> generate(int numRows) {  vector<vector<int>> res;  if(numRows<1) return res;  vector<int> lastRow(1,1);  res.push\_back(lastRow);  for(int i=1; i<numRows; i++)  {  vector<int> aRow(1,1);  for(int j=0; j<i-1;j++)  aRow.push\_back(lastRow[j] + lastRow[j+1]);  aRow.push\_back(1);  lastRow.swap(aRow);  res.push\_back(lastRow);  }  return res;  }  Java1 (1ms)  public List<List<Integer>> generate(int numRows) {  List<List<Integer>> res = new ArrayList<>();  for(int i=0; i<numRows; ++i)  {  int pre=0;  List<Integer> arow = new ArrayList<>();  for(int j=0; i>0&&j<res.get(i-1).size(); ++j)  {  arow.add(pre+res.get(i-1).get(j));  pre=res.get(i-1).get(j);  }  arow.add(1);  res.add(arow);  }  return res;  } |
| 119. Pascal's Triangle II  Given an index *k*, return the *k*th 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?  C++1 (3ms) O(k)space O(k2)time  vector<int> getRow(int rowIndex) {  vector<int>res(rowIndex+1, 1);  for(int i=2; i<=rowIndex; ++i)  {  int pre = 1;  for(int j=1; j<i; ++j)  {  res[j] += pre;  pre = res[j] - pre;  }  }  return res;  }  C++2 (0ms) O(k)space O(k2)time  vector<int> getRow(int rowIndex) {  vector<int> res;  for(int i=0; i<=rowIndex; i++)  {  res.push\_back(1);  for(int j=i-1; j>0; j--)  res[j] = res[j] + res[j-1];  }  return res;  }  C++3 (3ms) O(k2)space O(k2)time  vector<int> getRow(int rowIndex) {  vector<int> res(1,1);  for(int i=0; i<rowIndex; i++)  {  vector<int> alayer(1,1);  for(int j=0; j<i; j++)  alayer.push\_back(res[j] + res[j+1]);  alayer.push\_back(1);  res.swap(alayer);  }  return res;  }  C++4 (0ms) O(k)space O(k)time  vector<int> getRow(int rowIndex) {  vector<int> res(rowIndex+1,1);  for(int i=1; i<=(rowIndex+1)/2; i++)  {  res[i] = res[rowIndex-i] = (long)res[i-1]\*(long)(rowIndex-i+1)/i;  }  return res;  }  Java1 (4ms) O(k)space O(k2)time  public List<Integer> getRow(int rowIndex) {  List<Integer> res = new LinkedList<>();  res.add(1);  for(int i=0; i<rowIndex; ++i)  {  int pre=0;  for(int j=0; j<res.size(); ++j)  {  res.set(j, pre+res.get(j));  pre = res.get(j)-pre;  }  res.add(1);  }  return res;  } |
| 120. 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.  C++1  int minimumTotal(vector<vector<int>>& triangle) {  int h = triangle.size();  for(int i=h-2; i>=0; --i)  for(int j=0; j<triangle[i].size(); ++j)  triangle[i][j] += min(triangle[i+1][j], triangle[i+1][j+1]);  return triangle[0][0];  }  C++2  int minimumTotal(vector<vector<int>>& triangle) {  vector<int> minT = triangle.back();  for(int i=triangle.size()-2; i>=0; --i)  {  for(int j=0; j<triangle[i].size(); ++j)  {  minT[j] = triangle[i][j] + min(minT[j], minT[j+1]);  }  }  return minT[0];  } |
| 26. 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.  C++1 (29ms) O(n)  int removeDuplicates(vector<int>& nums) {  int i= !nums.empty();  for(int j=1; j<nums.size(); ++j)  {  if(nums[i-1] < nums[j])  nums[i++] = nums[j];  }  return i;  }  C++2 (32ms) O(n)  int removeDuplicates(vector<int>& nums) {  int len = nums.size();  if(len<2) return len;  int i=1;  for(int j=1;j<len; j++)  {  while(nums[i-1]==nums[j] && j<len) j++;  if(i==j)  {  i++;  continue;  }  if(j<len)  {  nums[i++] = nums[j];  }  }  return i;  }  C++3 (40ms) O(n)  int removeDuplicates(vector<int>& nums) {  int i = !nums.empty();  for (int n : nums)  if (n > nums[i-1])  nums[i++] = n;  return i;  }  Java1 (2ms)  public int removeDuplicates(int[] nums) {  int i=1, j=1, n=nums.length;  while(j<n)  {  if(nums[j]==nums[j-1])  {  j++;  continue;  }  nums[i++] = nums[j++];  }  return i;  }  Java2 (1ms)  public int removeDuplicates(int[] nums) {  int i=1, n=nums.length;  for(int j=1; j<n; ++j)  {  if(nums[j]!=nums[j-1]) nums[i++] = nums[j];  }  return i;  } |
| 303. Range Sum Query – Immutable  Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *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.   C++1 (206ms) write O(n) read O(1)  class NumArray {  private:  vector<int> sumLeft;  public:  NumArray(vector<int> nums) {  int n = nums.size();  sumLeft.resize(n+1,0);  int left=0;  for(int i=0; i<n; ++i)  {  sumLeft[i] = left;  left += nums[i];  }  sumLeft[n] = left;  }  int sumRange(int i, int j) {  return sumLeft[j+1] - sumLeft[i];  }  };  /\*\*  \* Your NumArray object will be instantiated and called as such:  \* NumArray obj = new NumArray(nums);  \* int param\_1 = obj.sumRange(i,j);  \*/  C++2 (28ms) write O(n) read O(1)  class NumArray {  public:  vector<int> numsM;  NumArray(vector<int> &nums) {  int n = nums.size();  numsM.resize(n+1);  for(int i=0; i<n; ++i)  numsM[i+1] = numsM[i] + nums[i];  }  int sumRange(int i, int j) {  return numsM[j+1] - numsM[i];  }  };  C++3 (28ms) write O(n) read O(1)  class NumArray {  private:  vector<int> numsM;  public:  NumArray(vector<int> &nums) {  numsM.push\_back(0);  for(int n : nums)  {  numsM.push\_back(numsM.back()+n);  }  }  int sumRange(int i, int j) {  return numsM[j+1] - numsM[i];  }  };  Java1 (3ms) write O(n) read O(1)  public class NumArray {  private int[] sums;  public NumArray(int[] nums) {  int n = nums.length;  sums = new int[n+1];  for(int i=0; i<n; ++i)  sums[i+1] = nums[i] + sums[i];  }  public int sumRange(int i, int j) {  return sums[j+1] - sums[i];  }  } |
| 307. Range Sum Query – Mutable  Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *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.   C++1 (TLE)  class NumArray {  private:  vector<int> nums;  vector<int> sums;  public:  NumArray(vector<int> nums) {  this->nums = nums;  sums = vector<int>(nums.size()+1);  for(int i=nums.size()-1; i>=0; --i)  sums[i] = nums[i] + sums[i+1];  }    void update(int i, int val) {  int change = val - nums[i];  nums[i] = val;  for(int k=i; k>=0; --k)  sums[k] += change;  }    int sumRange(int i, int j) {  return sums[i] - sums[j+1];  }  };  /\*\*  \* Your NumArray object will be instantiated and called as such:  \* NumArray obj = new NumArray(nums);  \* obj.update(i,val);  \* int param\_2 = obj.sumRange(i,j);  \*/  C++2  class NumArray {  private:  vector<int> BIT;  vector<int> Nums;  public:  NumArray(vector<int> &nums){  int n = nums.size();  Nums = nums;  //BIT = vector<int>(n+1);  BIT.resize(n+1);  for(int i=0; i<n; ++i)  {  updateBIT(i, nums[i]);  }  }  void update(int i, int val) {  if(val != Nums[i])  {  updateBIT(i, val-Nums[i]);  Nums[i] = val;  }  }  int sumRange(int i, int j) {  return leftSum(j) - leftSum(i-1);  }    int leftSum(int i)  {  i++;  int sum=0;  while(i>0)  {  sum += BIT[i];  i -= i & (-i);  }  return sum;  }    void updateBIT(int i, int val)  {  i++;  while(i<BIT.size())  {  BIT[i] +=val;  i += i & (-i); //double the last bit  }  }  }; |
| 453. 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]  C++1 (63ms)  int minMoves(vector<int>& nums) {  int minmum = INT\_MAX, sum=0;  for(int n : nums)  {  minmum = min(minmum, n);  sum += n;  }  return sum - nums.size()\*minmum;  }  C++2 (69ms)  int minMoves(vector<int>& nums) {  int minN = INT\_MAX;  int res = 0;  for(int num : nums)  minN = min(minN, num);    for(int num : nums)  res += num - minN;    return res;  } |
| 462. 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]  C++1 (19ms)  int minMoves2(vector<int>& nums) {  int median = nums.size()/2, res = 0;  nth\_element(nums.begin(), nums.begin()+median, nums.end());  for(int num : nums)  res += abs(num - nums[median]);  return res;  }  Java1 (14ms)  public int minMoves2(int[] nums) {  int res = 0;  Arrays.sort(nums);  int median = nums[nums.length/2];  for(int num : nums)  res += Math.abs(num- median);  return res;  } |
|  |
|  |
|  |
| 485. 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   C++1 (46ms) O(n)  int findMaxConsecutiveOnes(vector<int>& nums) {  int max1=0, cur=0;  for(int num : nums)  {  if(num==0) cur=0;  else  max1 = max(max1, ++cur);  }  return max1;  }  Python1  def findMaxConsecutiveOnes(self, nums):  res=0; count=0  for num in nums:  if num==0:  res=max(res, count)  count=0  else:  count += 1  return max(res, count) |
| 238. 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.)  C++1 (76ms) O(n)  vector<int> productExceptSelf(vector<int>& nums) {  int temp = 1, n=nums.size();  vector<int> res(n, 1);  for(int i=1; i<n; ++i)  res[i] = res[i-1]\*nums[i-1];  for(int i=n-2; i>=0; --i)  {  temp \*= nums[i+1];  res[i] = res[i] \* temp;  }  return res;  }  C++2 (64ms)  vector<int> productExceptSelf(vector<int>& nums) {  int fromBegin = 1, fromEnd = 1, len = nums.size();  vector<int> res(len, 1);  for(int i=0; i<len; ++i)  {  res[i] \*= fromBegin;  fromBegin \*= nums[i];  res[len-i-1] \*= fromEnd;  fromEnd \*= nums[len-i-1];  }  return res;  }  Java1 (4ms)  public int[] productExceptSelf(int[] nums) {  int len = nums.length;  int [] res = new int[len];  Arrays.fill(res, 1);  for(int i=1; i<len; ++i)  {  res[i] = res[i-1] \* nums[i-1];  }  int right=1;  for(int i=len-1; i>=0; --i)  {  res[i] \*= right;  right \*= nums[i];  }  return res;  }  Test cases:   |  |  |  | | --- | --- | --- | | Id | Testcase | Result | | 1 | 0,0,1,2 | 0,0,0,0 | | 2 | 0,1,2,3 | 6,0,0,0 | | 3 | 1,2,3 | 6,3,2 | | 4 | 0,0 | 0,0 | | 5 | 0,3 | 3,0 | | 6 | 3,0 | 0,3 | |
| 384. 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();  /\*\*  \* 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();  \*/  C++1 (232ms)  class Solution {  private:  vector<int> ori;  public:  Solution(vector<int> nums) {  ori = nums;  }    /\*\* Resets the array to its original configuration and return it. \*/  vector<int> reset() {  return ori;  }    /\*\* Returns a random shuffling of the array. \*/  vector<int> shuffle() {  vector<int> res = ori;  for(int i=1; i<res.size(); ++i)  swap(res[i], res[rand()%(i+1)]);  return res;  }  };  C++2  class Solution {  private:  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;  }  };  Java1  public class Solution {    private int[] oriArray;  private Random rand;  private int size;    public Solution(int[] nums) {  oriArray = nums;  size = nums.length;  rand = new Random();  }    /\*\* Resets the array to its original configuration and return it. \*/  public int[] reset() {  return oriArray;  }    /\*\* Returns a random shuffling of the array. \*/  public int[] shuffle() {  int[] shufArray = Arrays.copyOf(oriArray, size);  for(int i=1; i<size; ++i)  {  int r = rand.nextInt(i+1);  int temp = shufArray[i];  shufArray[i] = shufArray[r];  shufArray[r] = temp;  }  return shufArray;  }  }  Java2  public class Solution {    private int[] oriArray;  private Random rand;  private int size;    public Solution(int[] nums) {  oriArray = nums;  size = nums.length;  rand = new Random();  }    /\*\* Resets the array to its original configuration and return it. \*/  public int[] reset() {  return oriArray;  }    /\*\* Returns a random shuffling of the array. \*/  public int[] shuffle() {  int[] shufArray = Arrays.copyOf(oriArray, size);  for(int i=0; i<size; ++i)  {  int r = rand.nextInt(size - i);  int temp = shufArray[i];  shufArray[i] = shufArray[i+r];  shufArray[i+r] = temp;  }  return shufArray;  }  } |
|  |

#### Rotation

|  |
| --- |
| 189. 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]](https://leetcode.com/problems/rotate-array/?tab=Description)  **Hint:** Could you do it in-place with O(1) extra space?  Related problem: [Reverse Words in a String II](https://leetcode.com/problems/reverse-words-in-a-string-ii/)  C++1 (19ms) O(1)space O(n)  void rotate(vector<int>& nums, int k) {  k = k%nums.size();  reverse(nums.begin(), nums.end());  reverse(nums.begin(), nums.begin()+k);  reverse(nums.begin()+k, nums.end());  }  void rotate(vector<int>& nums, int k) {  reverse(nums.begin(), nums.end());  reverse(nums.begin(), nums.begin()+k%nums.size());  reverse(nums.begin()+k%nums.size(), nums.end());  }  C++2 (28ms) O(n)space O(n)  void rotate(vector<int>& nums, int k) {  int len = nums.size();  k = k%len;  if(len<2 || k==0) return;    // Make a copy of nums  vector<int> numsCopy(nums.begin(), nums.end());    // Rotate the elements.  for (int i = 0; i < len; i++)  {  nums[(i + k)%len] = numsCopy[i];  }  }  C++3 (44ms) O(1)space O(n)  void rotate(vector<int>& nums, int k) {  int len = nums.size();  k = k%len;  if(len<2 || k==0) return;  reverse(nums.begin(), nums.begin() + (len-k));  reverse(nums.begin()+ (len-k), nums.end());  reverse(nums.begin(), nums.end());  }  Java1 (0ms) O(n)space O(n)  public void rotate(int[] nums, int k) {  int n = nums.length;  k = k%n;  int[] temp = Arrays.copyOfRange(nums, 0, n-k);  System.arraycopy(nums, n-k, nums, 0, k);  System.arraycopy(temp, 0, nums, k, n-k);  } |
| Codility [CyclicRotation](https://codility.com/demo/results/training3MX52A-38H/)  Task description  A zero-indexed array A consisting of N integers is given. Rotation of the array means that each element is shifted right by one index, and the last element of the array is also moved to the first place.  For example, the rotation of array A = [3, 8, 9, 7, 6] is [6, 3, 8, 9, 7]. The goal is to rotate array A K times; that is, each element of A will be shifted to the right by K indexes.  Write a function:  vector<int> solution(vector<int> &A, int K);  that, given a zero-indexed array A consisting of N integers and an integer K, returns the array A rotated K times.  For example, given array A = [3, 8, 9, 7, 6] and K = 3, the function should return [9, 7, 6, 3, 8].  Assume that:   * N and K are integers within the range [0..100]; * each element of array A is an integer within the range [−1,000..1,000].   In your solution, focus on correctness. The performance of your solution will not be the focus of the assessment.  C++1  **vector**<**int**> solution(**vector**<**int**> &A, **int** K) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size();  **vector**<**int**> res(n);  **if**(n==0) **return** res;  K = n - (K%n);  **for**(**int** i=0; i<n; ++i)  res[i] = A[(i+K)%n];  **return** res;  }  TestCases  extreme\_empty  empty array  single  one element, 0 <= K <= 5  double  two elements, K <= N  small1  small functional tests, K < N  small2  small functional tests, K >= N  small\_random\_all\_rotations  small random sequence, all rotations, N = 15  medium\_random  medium random sequence, N = 100  maximal  maximal N and K |
| 396. Rotate Function  Given an array of integers A and let *n* to be its length.  Assume Bk 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 \* Bk[0] + 1 \* Bk[1] + ... + (n-1) \* Bk[n-1].  Calculate the maximum value of F(0), F(1), ..., F(n-1).  **Note:** *n* is guaranteed to be less than 105.  **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.  C++1  int maxRotateFunction(vector<int>& A) {  int sum=0, F=0, n=A.size();  for(int i=0; i<n; ++i)  {  sum += A[i];  F += i\*A[i];  }  int res = F;  for(int i=1; i<n; ++i)  {  F += sum - n\*A[n-i];  res = max(res, F);  }  return res;  } |
|  |

#### Duplicate

|  |
| --- |
| 287. 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(n2). 4. There is only one duplicate number in the array, but it could be repeated more than once.   C++1 (16ms) O(nlog(n))  int findDuplicate(vector<int>& nums) {  sort(nums.begin(), nums.end());  for(int i=1; i<nums.size(); ++i)  if(nums[i]==nums[i-1]) return nums[i];  return -1;  }  C++2 (16ms) O(n)  int findDuplicate(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]];  }  slow=0;  while(slow!=fast)  {  slow= nums[slow];  fast = nums[fast];  }  return slow;  }  return -1;  }  C++3 (24ms) O(log(n)n)  int findDuplicate(vector<int>& nums) {  if(nums.size()>1)  {  int i=1, j=nums.size()-1, mid, count;  while(i<j)  {  mid = i+(j-i)/2;  count=0;  for(int n : nums)  {  if(n<=mid) count++;  }  //the numsbers smaller than mid are less, it means the duplicate number must be larger than mid  if(count <= mid) i = mid+1;  else j = mid;  }  return i;  }  return -1;  } |
| 442. Find All Duplicates in an Array  Given an array of integers, 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear **twice** and others appear **once**.  Find all the elements that appear **twice** in this array.  Could you do it without extra space and in O(*n*) runtime?  **Example:**  **Input:**  [4,3,2,7,8,2,3,1]  **Output:**  [2,3]  C++1 (152ms)  vector<int> findDuplicates(vector<int>& nums) {  vector<int> res;  int n = nums.size();  for(int num : nums)  {  int id = (num-1+n)%n;  if(nums[id]<=0) res.push\_back(id+1);  else nums[id] -= n;  }  return res;  } |
| 448. Find All Numbers Disappeared in an Array  Given an array of integers where 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear twice and others appear once.  Find all the elements of [1, *n*] inclusive that do not appear in this array.  Could you do it without extra space and in O(*n*) runtime? You may assume the returned list does not count as extra space.  **Example:**  **Input:**  [4,3,2,7,8,2,3,1]  **Output:**  [5,6]  C++1 (142ms) O(n) O(1)space  vector<int> findDisappearedNumbers(vector<int>& nums) {  int n = nums.size();  for(auto num : nums)  {  int id = (num-1+n)%n;  if(nums[id] > 0)  nums[id] -= n;  }  vector<int> res;  for(int i=1; i<=n; ++i)  {  if(nums[i-1] > 0)  res.push\_back(i);  }  return res;  } |
|  |
| 349. 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.   C++1 (9ms)  vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {  set<int> set;  vector<int> res;  for(int num : nums1)  set.insert(num);  for(int num : nums2)  {  if(set.count(num)>0)  {  res.push\_back(num);  set.erase(num);  }  }  return res;  }  Java1 (6ms)  public int[] intersection(int[] nums1, int[] nums2) {  HashSet<Integer> sta1 = new HashSet();  HashSet<Integer> sta2 = new HashSet();  for(int n1 : nums1)  {  sta1.add(n1);  }  for(int n2 : nums2)  {  if(sta1.contains(n2)) sta2.add(n2);  }  int[] res = new int[sta2.size()];  int i=0;  for(Integer n : sta2) res[i++] = n;  return res;  } |
| 350. 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 *nums2*'s size? Which algorithm is better? * What if elements of *nums2* are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?   C++1 (9ms)  vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {  vector<int> res;  unordered\_map<int, int> counts;  for(int num1 : nums1)  counts[num1]++;  for(int num2 : nums2)  if(counts[num2]-- > 0)  res.push\_back(num2);  return res;  }  C++2 (6ms)  vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {  vector<int> res;  sort(nums1.begin(), nums1.end());  sort(nums2.begin(), nums2.end());  int i=0, j=0, len1=nums1.size(), len2=nums2.size();  while(i<len1 && j<len2)  {  if(nums1[i] == nums2[j])  {  res.push\_back(nums1[i]);  i++;  j++;  }  else if(nums1[i] > nums2[j]) j++;  else i++;  }  return res;  } |
| 217. Contains Duplicate  Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.  C++1 (48ms)  bool containsDuplicate(vector<int>& nums) {  unordered\_map<int, int> counts;  int len = nums.size();  for(int i=0; i<len; i++)  {  if(++counts[nums[i]]>1)  return true;  }  return false;  }  C++2 (66ms)  bool containsDuplicate(vector<int>& nums) {  if(nums.empty()) return false;  set<int> set;  for(int num : nums)  {  if(set.count(num)>0) return true;  set.insert(num);  }  return false;  }  Java1 (6ms)  public boolean containsDuplicate(int[] nums) {  Arrays.sort(nums);  for(int i=1; i<nums.length; ++i)  if(nums[i] == nums[i-1]) return true;  return false;  }  Java2 (9ms)  public boolean containsDuplicate(int[] nums) {  Set<Integer> set = new HashSet();  for(int num : nums)  if(!set.add(num)) return true;  return false;  } |
| 219. 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 **absolute** difference between *i* and *j* is at most *k*.  C++1 (32ms)  bool containsNearbyDuplicate(vector<int>& nums, int k) {  unordered\_map<int, int> count;  for(int i=0; i<nums.size(); i++)  {  if(count[nums[i]] && i - count[nums[i]] < k) return true;  count[nums[i]] = i+1;  }  return false;  }  Java1 (19ms)  public boolean containsNearbyDuplicate(int[] nums, int k) {  Map<Integer, Integer> match = new HashMap<Integer, Integer>();  for(int i=0; i<nums.length; ++i)  {  if(match.containsKey(nums[i]) && (i-match.get(nums[i])) < k)  return true;  match.put(nums[i], i+1);  }  return false;  } |
| Codility [OddOccurrencesInArray](https://codility.com/demo/results/trainingPZV86Q-9H5/)  A non-empty zero-indexed array A consisting of N integers is given. The array contains an odd number of elements, and each element of the array can be paired with another element that has the same value, except for one element that is left unpaired.  For example, in array A such that:  A[0] = 9 A[1] = 3 A[2] = 9 A[3] = 3 A[4] = 9 A[5] = 7 A[6] = 9   * the elements at indexes 0 and 2 have value 9, * the elements at indexes 1 and 3 have value 3, * the elements at indexes 4 and 6 have value 9, * the element at index 5 has value 7 and is unpaired.   Write a function:  int solution(vector<int> &A);  that, given an array A consisting of N integers fulfilling the above conditions, returns the value of the unpaired element.  For example, given array A such that:  A[0] = 9 A[1] = 3 A[2] = 9 A[3] = 3 A[4] = 9 A[5] = 7 A[6] = 9  the function should return 7, as explained in the example above.  Assume that:   * N is an odd integer within the range [1..1,000,000]; * each element of array A is an integer within the range [1..1,000,000,000]; * all but one of the values in A occur an even number of times.   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **for**(**int** i=1; i<A.size(); ++i)  A[0] ^= A[i];  **return** A[0];  } |
| 80. 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.  C++1  int removeDuplicates(vector<int>& nums) {  int n = nums.size();  int i=0;  for(int j=0; j<n; ++i, ++j)  {  if(i==0) continue;  while(nums[j]==nums[j-1] &&(j+1<n && nums[j]==nums[j+1])) j++;  nums[i] = nums[j];  }  return i;  }  C++2  int removeDuplicates(vector<int>& nums) {  int i = 0;  for (int n : nums)  if (i < 2 || n > nums[i-2])  nums[i++] = n;  return i;  } |
| 645. Set Mismatch  The set S originally contains numbers from 1 to n. But unfortunately, due to the data error, one of the numbers in the set got duplicated to **another** number in the set, which results in repetition of one number and loss of another number.  Given an array nums representing the data status of this set after the error. Your task is to firstly find the number occurs twice and then find the number that is missing. Return them in the form of an array.  **Example 1:**  **Input:** nums = [1,2,2,4]  **Output:** [2,3]  **Note:**   1. The given array size will in the range [2, 10000]. 2. The given array's numbers won't have any order.   C++1  vector<int> findErrorNums(vector<int>& nums) {  int n = nums.size();  vector<int> res(2), count(n);  for(int num : nums)  count[num-1]++;  for(int i=1; i<=n; ++i)  {  if(count[i-1]==2)  res[0] = i;  else if(count[i-1]==0)  res[1] = i;  }  return res;  }  Java1  public int[] findErrorNums(int[] nums) {  int n = nums.length;  int[] count = new int[n];  int[] res = new int[2];  for(int num : nums){  count[num-1]++;  if(count[num-1]==2)  res[0] = num;  }  for(int i=0; i<n; ++i)  if(count[i]==0){  res[1] = i+1;  break;  }  return res;  } |
|  |

#### Permutations

|  |
| --- |
| 46. 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],  [3,2,1]  ]  C++1 (13ms) O(n2)  class Solution {  public:  void permutaion(vector<vector<int>>& res, vector<int>& aRow, vector<int>& nums, int start)  {  if(start == nums.size()) res.push\_back(aRow);  else  {  for(int i=start; i<nums.size(); ++i)  {  swap(nums[start], nums[i]);  aRow.push\_back(nums[start]);  permutaion(res, aRow, nums, start+1);  swap(nums[start], nums[i]);  aRow.pop\_back();  }  }  }  vector<vector<int>> permute(vector<int>& nums) {  vector<vector<int>> res;  vector<int> aRow;  permutaion(res, aRow, nums, 0);  return res;  }  };  C++2 (16ms)  class Solution {  public:  void permuteHelper(vector<vector<int>>& res, vector<int>& aRow, vector<int>& nums)  {  if(aRow.size() == nums.size())  res.push\_back(aRow);  else  {  for(int i=0; i<nums.size(); ++i)  {  if(find(aRow.begin(), aRow.end(), nums[i]) == aRow.end())  {  aRow.push\_back(nums[i]);  permuteHelper(res, aRow, nums);  aRow.pop\_back();  }  }  }  }  vector<vector<int>> permute(vector<int>& nums) {  vector<vector<int>> res;  vector<int> aRow;  permuteHelper(res, aRow, nums);  return res;  }  };  C++3  class Solution {  public:  void permutaion(vector<vector<int>>& res,vector<int>& nums, int start)  {  if(start == nums.size()) res.push\_back(nums);  else  {  for(int i=start; i<nums.size(); ++i)  {  swap(nums[start], nums[i]);  permutaion(res, nums, start+1);  swap(nums[start], nums[i]);  }  }  }  vector<vector<int>> permute(vector<int>& nums) {  vector<vector<int>> res;  permutaion(res, nums, 0);  return res;  }  };  Java1 (6ms)  public class Solution {  private List<List<Integer>> res;    public List<List<Integer>> permute(int[] nums) {  res = new ArrayList<>();  List<Integer> arow = new LinkedList<Integer>();  createPermute(arow, nums, 0);  return res;  }    private void createPermute(List<Integer> arow, int[] nums, int start)  {  if(arow.size() == nums.length)  {  List<Integer> ans = new LinkedList<Integer>(arow);  res.add(ans);  return;  }  for(int i=start; i<nums.length; ++i)  {  mySwap(nums, start, i);  arow.add(nums[start]);  createPermute(arow, nums, start+1);  mySwap(nums, start, i);  arow.remove(arow.size()-1);  }  }    private void mySwap(int[] nums, int i, int j)  {  int temp = nums[i];  nums[i] = nums[j];  nums[j] = temp;  }  } |
| 47. 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],  [2,1,1]  ]  C++1  class Solution {  public:  void permuteDFS(vector<vector<int>>& res, vector<int> nums, int start)  {  if(start==nums.size()-1)  {  res.push\_back(nums);  return;  }  for(int i=start; i<nums.size(); ++i)  {  if(i!=start && nums[i]==nums[start]) continue;  swap(nums[i], nums[start]);  permuteDFS(res, nums, start+1);  }  }  vector<vector<int>> permuteUnique(vector<int>& nums) {  vector<vector<int>> res;  sort(nums.begin(), nums.end());  permuteDFS(res, nums, 0);  return res;  }  }; |
| Codility  [PermCheck](https://codility.com/demo/results/trainingSGJ754-KND/#task-0)  A non-empty zero-indexed array A consisting of N integers is given.  A *permutation* is a sequence containing each element from 1 to N once, and only once.  For example, array A such that:  A[0] = 4 A[1] = 1 A[2] = 3 A[3] = 2  is a permutation, but array A such that:  A[0] = 4 A[1] = 1 A[2] = 3  is not a permutation, because value 2 is missing.  The goal is to check whether array A is a permutation.  Write a function:  int solution(vector<int> &A);  that, given a zero-indexed array A, returns 1 if array A is a permutation and 0 if it is not.  For example, given array A such that:  A[0] = 4 A[1] = 1 A[2] = 3 A[3] = 2  the function should return 1.  Given array A such that:  A[0] = 4 A[1] = 1 A[2] = 3  the function should return 0.  Assume that:   * N is an integer within the range [1..100,000]; * each element of array A is an integer within the range [1..1,000,000,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size();  **vector**<**bool**> bucket(n+1);  **for**(**int** a : A)  **if**(a>0 && a<=n)  bucket[a] = **true**;  **for**(**int** i=1; i<=n; ++i)  **if**(bucket[i] == **false**)  **return** 0;  **return** 1;  }  extreme\_min\_max  single element with minimal/maximal value  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  single  single element  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  double  two elements  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  antiSum1  total sum is correct, but it is not a permutation, N <= 10  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  small\_permutation  permutation + one element occurs twice, N = ~100  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  medium\_permutation  permutation + few elements occur twice, N = ~10,000  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  antiSum2  total sum is correct, but it is not a permutation, N = ~100,000  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  large\_permutation  permutation + one element occurs three times, N = ~100,000  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  large\_range  sequence 1, 2, ..., N, N = ~100,000  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  extreme\_values  all the same values, N = ~100,000 |
| 31. 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  C++1  void nextPermutation(vector<int>& nums) {  int n = nums.size();  int i=n-2;  while(i>=0 && nums[i]>=nums[i+1])  i--;  if(i==-1)  {  reverse(nums.begin(), nums.end());  return;  }  int j = i+1;  for(; j<n-1; ++j) //O(n)  if(nums[i]<nums[j] && nums[i]>=nums[j+1])  break;  swap(nums[i], nums[j]);  reverse(nums.begin()+i+1, nums.end());  }  C++2  void nextPermutation(vector<int>& nums) {  if(nums.empty()) return;  int i=nums.size()-2;  while(i>=0 && nums[i]>=nums[i+1]) i--;  reverse(nums.begin()+i+1,nums.end());  if(i==-1) return;  auto it = upper\_bound(nums.begin()+i+1,nums.end(), nums[i]); //O(log(n))  swap(nums[i], \*it);  } |
| 496. 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 array 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 array 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.   C++1 (9ms) O(mn)  class Solution {  public:  int nextGreater(vector<int>& nums, int start, int val)  {  for(int i=start; i<nums.size(); ++i)  if(nums[i]>val) return nums[i];  return -1;  }  vector<int> nextGreaterElement(vector<int>& findNums, vector<int>& nums) {  vector<int>res(findNums.size());  unordered\_map<int, int> addr;  for(int i=0; i<nums.size(); ++i)  addr[nums[i]] = i+1;  for(int i=0; i<findNums.size(); ++i)  res[i] = nextGreater(nums, addr[findNums[i]], findNums[i]);  return res;  }  };  Python1  def nextGreaterElement(self, findNums, nums):  addr = dict([(nums[i],i) for i in range(len(nums))])  return [self.getNextGreater(addr[findNums[i]], nums) for i in range(len(findNums))]    def getNextGreater(self, f, nums):  for i in range(f+1, len(nums)):  if nums[i] > nums[f]:  return nums[i]  return -1 |
|  |
| 556. Next Greater Element III  Given a positive **32-bit** integer **n**, you need to find the smallest **32-bit** integer which has exactly the same digits existing in the integer **n** and is greater in value than n. If no such positive **32-bit** integer exists, you need to return -1.  **Example 1:**  **Input:** 12  **Output:** 21  **Example 2:**  **Input:** 21  **Output:** -1  C++1  class Solution {  public:  void reverse(string& s, int left)  {  int right = s.length()-1;  while(left < right)  swap(s[left++], s[right--]);  }  int nextGreaterElement(int n) {  string s = to\_string(n);  int i = s.length()-2;  while(i>=0 && s[i]>=s[i+1]) i--;  if(i==-1) return -1;  int j=s.length()-1;  while(s[i] >= s[j]) j--;  swap(s[i], s[j]);  reverse(s, i+1);  try{  return stoi(s);  }  catch(exception e){  return -1;  }  //long res = stol(s);  //return res>INT\_MAX ? -1 : res;  }  }; |
| 60. 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 *k*th permutation sequence.  **Note:** Given *n* will be between 1 and 9 inclusive.  C++1  string getPermutation(int n, int k) {  vector<char> digits(n);  vector<int>factorials(n+1);  factorials[0] = 1;  int fSum = 1;  for(int i=1; i<=n; ++i)  {  digits[i-1] = '0'+i; //digits[n] = {1,2,3,..., n};  fSum \*= i;  factorials[i] = fSum; //int fac[9] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320};  }  string res;  k-=1;  for(int i=n-1; i>=0; --i)  {  int th = k/factorials[i];  res += digits[th];  digits.erase(digits.begin()+th);  k %= factorials[i];  }  return res;  }  C++2  string getPermutation(int n, int k) {  vector<char> digit;  for(int i=1; i<=n; ++i) digit.push\_back('0'+i);  int fac[9] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320};  string res="";  k -= 1;  for(int i=n; i>=1; --i)  {  int th = k/fac[i-1];  res += digit[th];  digit.erase(digit.begin()+th);  k %= fac[i-1];  }  return res;  }  C++3  class Solution {  public:  char getThEmpty(vector<bool>& digit, int th)  {  int cnt =0;  for(int i=0; i<digit.size(); ++i)  {  if(digit[i]==false)  {  if(cnt==th)  {  digit[i] = true;  return '0'+i+1;  }  cnt++;  }  }  return '0';  }  string getPermutation(int n, int k) {  vector<bool> digit(n, false);  int fac[8] = {1, 2, 6, 24, 120, 720, 5040, 40320};  string res="";  for(int i=n-1; i>=1; --i)  {  int th = (k-1)/fac[i-1];  res += getThEmpty(digit, th);  k %= fac[i-1];  if(k==0) break;  }  for(int i=n-1; i>=0; --i)  {  if(digit[i]==false)  {  digit[i]==true;  res += '0'+i+1;  }  }  return res;  }  }; |
|  |
|  |

#### Stock buy and sell

|  |
| --- |
| 121. Best Time to Buy and Sell Stock  Say you have an array for which the *i*th 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.  **Example 1:**  Input: [7, 1, 5, 3, 6, 4]  Output: 5  max. difference = 6-1 = 5 (not 7-1 = 6, as selling price needs to be larger than buying price)  **Example 2:**  Input: [7, 6, 4, 3, 1]  Output: 0  In this case, no transaction is done, i.e. max profit = 0.  C++1 (8ms)  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;  }  Java1 (2ms)  public int maxProfit(int[] prices) {  int buy=Integer.MIN\_VALUE, sell=0;  for(int price : prices)  {  sell = Math.max(sell, buy+price);  buy = Math.max(buy, -price);  }  return sell;  }  Testcase  simple\_desc  descending and ascending sequence, length=5  [▶](https://codility.com/demo/results/training9H53RY-NUR/)  simple\_empty  empty and [0,200000] sequence  [▶](https://codility.com/demo/results/training9H53RY-NUR/)  two\_hills  two increasing subsequences  [▶](https://codility.com/demo/results/training9H53RY-NUR/)  max\_profit\_after\_max\_and\_before\_min  max profit is after global maximum and before global minimum  [▶](https://codility.com/demo/results/training9H53RY-NUR/)  medium\_1  large value (99) followed by short V-pattern (values from [1..5]) repeated 100 times  [▶](https://codility.com/demo/results/training9H53RY-NUR/)  large\_1  large value (99) followed by short pattern (values from [1..6]) repeated 10K times  [▶](https://codility.com/demo/results/training9H53RY-NUR/)  large\_2  chaotic sequence of 200K values from [100K..120K], then 200K values from [0..100K]  [▶](https://codility.com/demo/results/training9H53RY-NUR/)  large\_3  chaotic sequence of 200K values from [1..200K] |
| 122. Best Time to Buy and Sell Stock II  Say you have an array for which the *i*th 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).  C++1  int maxProfit(vector<int>& prices) {  int res = 0;  for(int i=1; i<prices.size(); ++i)  res += max(0, prices[i] - prices[i-1]);  return res;  } |
| 309. Best Time to Buy and Sell Stock with Cooldown  Say you have an array for which the *i*th 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]  C++1  int maxProfit(vector<int>& prices) {  int buy(INT\_MIN), sell(0), prev\_buy(0), prev\_sell(0);  //int preBuy=0, preSell=0, buy=INT\_MIN, sell=0;  for(int price : prices)  {  prev\_buy = buy;  buy = max(prev\_sell - price, prev\_buy);  prev\_sell = sell;  sell = max(prev\_buy + price, prev\_sell);  }  return sell;  }  Java1  public int maxProfit(int[] prices) {  int buy = Integer.MIN\_VALUE;  int sell=0, preBuy=0, preSell=0;  for(int price : prices)  {  preBuy = buy;  buy = Math.max(preSell-price, preBuy);  preSell = sell;  sell = Math.max(preBuy+price, preSell);  }  return sell;  } |
|  |
|  |

#### SubArray

|  |
| --- |
| 53. 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.](https://leetcode.com/problems/maximum-subarray/)  **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.  C++1 (12ms)  int maxSubArray(vector<int>& nums) {  int preMax = nums[0], curMax = nums[0];  for(int i=1; i<nums.size(); ++i)  {  preMax = max(preMax+nums[i], nums[i]);  curMax = max(preMax, curMax); //save the maximum sum  }  return curMax;  }  C++2 (15ms)  int maxSubArray(vector<int>& nums) {  int maxCur=0, maxSub=INT\_MIN;  for(int num : nums)  {  maxCur = max(maxCur + num, num);  maxSub = max(maxSub, maxCur);  }  return maxSub;  } |
| 300. 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(*n2*) complexity.  **Follow up:** Could you improve it to O(*n* log *n*) time complexity?  C++1(6ms) set  int lengthOfLIS(vector<int>& nums) {  set<int> lis;  for(auto num : nums)  {  if(lis.find(num) == lis.end())  {  lis.insert(num);  auto it = lis.upper\_bound(num);  if(it != lis.end())  lis.erase(it);  }  }  return lis.size();  }  Java1 (1ms) using Array as stack  public int lengthOfLIS(int[] nums) {  int len = nums.length;  int[] stack = new int[len];  int top =-1;  for(int num : nums)  {  int cur = top;  while(cur!=-1 && stack[cur]>=num)  cur--;  if(cur == top)  top++;  stack[++cur] = num; //replace upper bound  }  return top + 1;  }  Java2  public int lengthOfLIS(int[] nums) {  int len = nums.length;  int[] dp = new int[len];  int size = 0;  for(int num : nums)  {  int i = Arrays.binarySearch(dp, 0, size, num);  if(i<0) i = -(i+1);  dp[i] = num;  if(i==size) size++;  }  return size;  }  Java3  public int lengthOfLIS(int[] nums) {  LinkedList<Integer> dp = new LinkedList<Integer>();  for(int num : nums)  {  if(dp.isEmpty() || dp.getLast() < num)  dp.add(num);  else  {  int i = Collections.binarySearch(dp, num);  dp.set((i<0)? -(i+1) : i, num);  }  }  return dp.size();  } |
| 491. 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.   C++1 (296ms)  class Solution {  public:  void backTracking(vector<vector<int>>& res, vector<int>& sub, vector<int>& nums, int start, int end)  {  if(sub.size()>=2) res.push\_back(sub);  unordered\_map<int, int> map;  for(int i=start; i<end; ++i)  {  if(map[nums[i]] > 0) continue;  if(sub.size()==0 || nums[i]>=sub.back())  {  map[nums[i]]++;  sub.push\_back(nums[i]);  backTracking(res, sub, nums, i+1, end);  sub.pop\_back();  }  }  }  vector<vector<int>> findSubsequences(vector<int>& nums) {  vector<vector<int>> res;  vector<int> sub;  backTracking(res, sub, nums, 0, nums.size());  return res;  }  }; |
| 523. 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.   C++1 (95ms)  bool checkSubarraySum(vector<int>& nums, int k) {  vector<int> sumNums(nums.begin(), nums.end());  int i=0, j=1;  while(i+j < nums.size())  {  for(;i+j<nums.size(); ++i)  {  sumNums[i] += nums[i+j];  if(k==0 && sumNums[i]==0) return true;  if(k!=0 && sumNums[i]%k == 0) return true;  }  i = 0;  ++j;  }  return false;  }  C++2 (56ms)  bool checkSubarraySum(vector<int>& nums, int k) {  vector<int> sumNums(nums.begin(), nums.end());  int i=0, j=1;  while(i+j < nums.size())  {  for(;i+j<nums.size(); ++i)  {  sumNums[i] += nums[i+j];  if(k==0)  {  if(sumNums[i]==0) return true;  }  else if(sumNums[i]%k == 0) return true;  }  i = 0;  ++j;  }  return false;  }  C++3 (56ms)  bool checkSubarraySum(vector<int>& nums, int k) {  vector<int> sumNums(nums.begin(), nums.end());    for(int j=1; j<nums.size(); ++j)  {  for(int i=0;i+j<nums.size(); ++i)  {  sumNums[i] += nums[i+j];  if(k==0)  {  if(sumNums[i]==0) return true;  }  else if(sumNums[i]%k == 0) return true;  }  }  return false;  } |
| 561. Array Partition I  Given an array of **2n** integers, your task is to group these integers into **n** pairs of integer, say (a1, b1), (a2, b2), ..., (an, bn) which makes sum of min(ai, bi) for all i from 1 to n as large as possible.  **Example 1:**  **Input:** [1,4,3,2]  **Output:** 4  **Explanation:** n is 2, and the maximum sum of pairs is 4.  **Note:**   1. **n** is a positive integer, which is in the range of [1, 10000]. 2. All the integers in the array will be in the range of [-10000, 10000].   C++1  int arrayPairSum(vector<int>& nums) {  int res = 0;  sort(nums.begin(), nums.end());  for(int i=0; i<nums.size(); ++i)  if(i % 2 == 0)  res += nums[i];  return res;  } |
| 376. 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?  C++1  int wiggleMaxLength(vector<int>& nums) {  int count=1, sign=0;  for(int i=1; i<nums.size(); ++i)  {  if(nums[i]>nums[i-1] && sign!=1)  {  count++;  sign = 1;  }  else if(nums[i]<nums[i-1] && sign!=-1)  {  count++;  sign = -1;  }  }  return nums.size()==0 ? 0 : count;  } |
|  |
| 324. Wiggle Sort II  Given an unsorted array nums, reorder it such that nums[0] < nums[1] > nums[2] < nums[3]....  **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?  C++1  void wiggleSort(vector<int>& nums) {  vector<int> sorted(nums);  sort(sorted.begin(), sorted.end());  for(int i = nums.size()-1, j=0, mid=i/2+1; i>=0; --i)  nums[i] = sorted[i&1 ? mid++ : j++];  }  C++2  void wiggleSort(vector<int>& nums) {  int n = nums.size();  nth\_element(nums.begin(), nums.begin()+n/2, nums.end());  int mid = nums[n/2];    #define A(i) nums[(2\*i+1)% (n|1)]    int left=0, right=n-1, j=0;  while(j<=right)  {  if(A(j) > mid)  swap(A(left++), A(j++));  else if(A(j) < mid)  swap(A(j), A(right--));  else  j++;  }  } |
| 78. Subsets  Given a set of **distinct** integers, *nums*, return all possible subsets.  **Note:** 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],  []  ]  C++1 (9ms)  class Solution {  public:  void backTracking(vector<vector<int>>& res, vector<int>& aRow, vector<int>& nums, int start, int end)  {  if(start <= end)  {  res.push\_back(aRow);  for(int i=start; i<end; ++i)  {  aRow.push\_back(nums[i]);  backTracking(res, aRow, nums, i+1, end);  aRow.pop\_back();  }  }  }  vector<vector<int>> subsets(vector<int>& nums) {  vector<vector<int>> res;  vector<int> aRow;  backTracking(res, aRow, nums, 0, nums.size());  return res;  }  };  C++2  vector<vector<int>> subsets(vector<int>& nums) {  vector<vector<int>> res(1,vector<int>());  for(int i=0; i<nums.size(); ++i)  {  int n = res.size();  for(int j=0; j<n; ++j)  {  res.push\_back(res[j]);  res.back().push\_back(nums[i]);  }  }  return res;  }  Java1  public List<List<Integer>> subsets(int[] nums) {  List<List<Integer>> res = new ArrayList<>();  List<Integer> firstRow = new ArrayList<Integer>();  res.add(firstRow);  //Arrays.sort(nums);  for(int i=0; i<nums.length; ++i){  int len = res.size();  for(int j=0; j<len; ++j){  res.add(new ArrayList<Integer>(res.get(j)));  res.get(res.size()-1).add(nums[i]);  }  }  return res;  }  Java2  public class Solution {    private List<List<Integer>> res;    public List<List<Integer>> subsets(int[] nums) {  res = new LinkedList<>();  List<Integer> arow = new LinkedList<Integer>();  collectSubsets(arow, nums, 0);  return res;  }    private void collectSubsets(List<Integer> arow, int[] nums, int start){  List<Integer> ares = new LinkedList<Integer>(arow);  res.add(ares);  for(int i=start; i<nums.length; ++i){  arow.add(nums[i]);  collectSubsets(arow, nums, i+1);  arow.remove(arow.size()-1);  }  }  } |
| 90. Subsets II  Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets.  **Note:** 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],  []  ]  C++1 (back tracking)  class Solution {  public:  void CollectSubsets(vector<vector<int>>& res, vector<int> aRow, int start, vector<int> nums)  {  if(start >= nums.size()) return;  for(int i=start; i<nums.size(); ++i)  {  if(i!=start && nums[i]==nums[i-1]) continue;  aRow.push\_back(nums[i]);  res.push\_back(aRow);  CollectSubsets(res, aRow, i+1, nums);  aRow.pop\_back();  }  }  vector<vector<int>> subsetsWithDup(vector<int>& nums) {  vector<vector<int>> res;  vector<int> aRow;  res.push\_back(aRow);  sort(nums.begin(), nums.end());  CollectSubsets(res, aRow, 0, nums);  return res;  }  };  C++2 (iteration)  vector<vector<int>> subsetsWithDup(vector<int>& nums) {  sort(nums.begin(), nums.end());  vector<vector<int>> res(1, vector<int>());  vector<int> aRow;  int size = 1;  for(int i=0; i<nums.size(); ++i)  {  int start = (i>0)&&(nums[i]==nums[i-1]) ? size : 0;  size = res.size();  for(int j=start; j<size; ++j)  {  res.push\_back(res[j]);  res.back().push\_back(nums[i]);  }  }  return res;  } |
| 494. 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.   C++1 (386ms)  class Solution {  public:  int TargetSum(vector<int>& nums, int S, int i)  {  if(i == nums.size())  return S==0 ? 1 : 0;  int res = 0;  res += TargetSum(nums, S+nums[i], i+1);  res += TargetSum(nums, S-nums[i], i+1);  return res;  }  int findTargetSumWays(vector<int>& nums, int S) {  return TargetSum(nums, S, 0);  }  };  C++2 (3ms) DP  class Solution {  public:  int subsetSum(vector<int>& nums, int target)  {  vector<int>dp(target+1);  dp[0]=1;  for(auto num : nums)  for(int i=target; i>=num; --i)  dp[i] += dp[i-num];  return dp[target];  }  int findTargetSumWays(vector<int>& nums, int S) {  int sum = accumulate(nums.begin(), nums.end(), 0);  return (sum<S || (sum+S)%2 != 0) ? 0 : subsetSum(nums, (sum+S)/2);  }  }; |
|  |
|  |

##### Continuous Subarray

|  |
| --- |
| 581. Shortest Unsorted Continuous Subarray  Given an integer array, you need to find one **continuous subarray** that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order, too.  You need to find the **shortest** such subarray and output its length.  **Example 1:**  **Input:** [2, 6, 4, 8, 10, 9, 15]  **Output:** 5  **Explanation:** You need to sort [6, 4, 8, 10, 9] in ascending order to make the whole array sorted in ascending order.  **Note:**   1. Then length of the input array is in range [1, 10,000]. 2. The input array may contain duplicates, so ascending order here means **<=**.   C++1  int findUnsortedSubarray(vector<int>& nums) {  vector<int> sorted(nums);  sort(sorted.begin(), sorted.end());  int i=0, n=nums.size();  for(; i<n; ++i)  if(nums[i]!=sorted[i]) break;  int j=n-1;  for(; j>i; --j)  if(nums[j]!=sorted[j]) break;  return i==j ? 0 : j-i+1;  } |
| 643. Maximum Average Subarray I  Given an array consisting of n integers, find the contiguous subarray of given length k that has the maximum average value. And you need to output the maximum average value.  **Example 1:**  **Input:** [1,12,-5,-6,50,3], k = 4  **Output:** 12.75  **Explanation:** Maximum average is (12-5-6+50)/4 = 51/4 = 12.75  **Note:**   1. 1 <= k <= n <= 30,000. 2. Elements of the given array will be in the range [-10,000, 10,000].   C++1  double findMaxAverage(vector<int>& nums, int k) {  double res=0, sum=0;  for(int i=0; i<k; ++i)  sum += nums[i];  res = sum;  for(int i=0, j=k; j<nums.size(); i++, j++)  {  sum = sum - nums[i] + nums[j];  res = max(res, sum);  }  return res/k;  }  Java1  public double findMaxAverage(int[] nums, int k) {  double res=0, sum=0;  for(int i=0; i<k; ++i)  sum += nums[i];  res = sum;  for(int i=0, j=k; j<nums.length; ++i, ++j){  sum = sum - nums[i] + nums[j];  res = Math.max(res, sum);  }  return res/k;  } |
|  |
|  |

##### Uncontinuous

|  |
| --- |
| 594. Longest Harmonious Subsequence  We define a harmonious array is an array where the difference between its maximum value and its minimum value is **exactly** 1.  Now, given an integer array, you need to find the length of its longest harmonious subsequence among all its possible [subsequences](https://en.wikipedia.org/wiki/Subsequence).  **Example 1:**  **Input:** [1,3,2,2,5,2,3,7]  **Output:** 5  **Explanation:** The longest harmonious subsequence is [3,2,2,2,3].  **Note:** The length of the input array will not exceed 20,000.  C++1  int findLHS(vector<int>& nums) {  map<int, int> count;  int res=0;  for(int num : nums)  count[num]++;  for(auto c : count)  if(count.count(c.first + 1) > 0)  res = max(res, c.second + count[c.first + 1]);  return res;  }  C++2  int findLHS(vector<int>& nums) {  map<int, int> count;  int res=0;  for(int num : nums)  count[num]++;  for(auto c = count.begin(); c!=count.end(); ++c)  if(count.count(c->first + 1) > 0)  res = max(res, c->second + count[c->first + 1]);  return res;  } |
|  |
|  |

#### Move elements

|  |
| --- |
| 283. 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:**  C++1  void moveZeroes(vector<int>& nums) {  for(int j=0, i=0; j<nums.size(); ++j)  {  if(nums[j]!=0)  {  if(i!=j)  {  nums[i] = nums[j];  nums[j] = 0;  }  i++;  }  }  }  Java1  public void moveZeroes(int[] nums) {  int slow=0, fast=0, n=nums.length;  while(fast<n)  {  while(fast<n-1 && nums[fast]==0) fast++;  if(slow==fast)  {  slow++;  fast++;  continue;  }  nums[slow++] = nums[fast];  nums[fast++] = 0;  }  } |
|  |
|  |
|  |

#### Traversal

|  |
| --- |
| 498. 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:**  https://leetcode.com/static/images/problemset/diagonal_traverse.png  **Note:**   1. The total number of elements of the given matrix will not exceed 10,000.   C++1 (88ms)  vector<int> findDiagonalOrder(vector<vector<int>>& matrix) {  int h = matrix.size();  int w = (h>0) ? matrix[0].size() : 0;  vector<int> res(h\*w);  int id=0;  for(int i=0; i<h+w; ++i)  {  int lb = max(0, i-w+1), ub = 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;  } |
| 59. 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 ]  ]  C++1 (3ms)  vector<vector<int>> generateMatrix(int n) {  vector<vector<int>> res(n, vector<int>(n));  int val=0, i=0, j=-1;  while(val<n\*n)  {  while(j+1<n && res[i][j+1]==0) res[i][++j] = ++val;  while(i+1<n && res[i+1][j]==0) res[++i][j] = ++val;  while(j-1>=0 && res[i][j-1]==0) res[i][--j] = ++val;  while(i-1>=0 && res[i-1][j]==0) res[--i][j] = ++val;  }  return res;  } |
| 503. 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 greater 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.  C++1 (385ms) O(n2)  vector<int> nextGreaterElements(vector<int>& nums) {  int n = nums.size();  vector<int>res(n, INT\_MIN);  for(int i=0; i<n; ++i)  {  for(int j=i+1; j<=n+i; ++j)  {  int next = j%n;  if(nums[next] > nums[i])  {  res[i] = nums[next];  break;  }  }  if(res[i]==INT\_MIN) res[i] = -1;  }  return res;  }  C++2 (113ms) O(n)  vector<int> nextGreaterElements(vector<int>& nums) {  int n = nums.size();  vector<int>res(n, -1);  stack<int> sta;  for(int i=n-1; i>=0; --i)  sta.push(i);  for(int i=n-1; i>=0; --i)  {  while(!sta.empty() && nums[sta.top()]<= nums[i])  sta.pop();  if(!sta.empty()) res[i] = nums[sta.top()];  sta.push(i);  }  return res;  } |
| 334. 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 ≤ *i* < *j* < *k* ≤ *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.  C++1 (6ms)  bool increasingTriplet(vector<int>& nums) {  vector<int> res(2, INT\_MAX);  for(auto num : nums)  {  if(num <= res[0]) res[0] = num;  else if(num <= res[1]) res[1] = num;  else return true;  }  return false;  }  C++2 (12ms)  bool increasingTriplet(vector<int>& nums) {  if(nums.size()<3) return false;  set<int> s;  for(int n : nums)  {  if(s.find(n) != s.end()) continue;  s.insert(n);  auto it = s.upper\_bound(n);  if(it != s.end()) s.erase(it);  if(s.size() == 3) return true;  }  return false;  } |
| 539. 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.   C++1  class Solution {  public:  int getTime(string timePoint)  {  int i=0;  while(timePoint[i]!=':') i++;  int hour = stoi(timePoint.substr(0,i));  int mins = stoi(timePoint.substr(i+1));  return 60\*hour + mins;  }  int findMinDifference(vector<string>& timePoints) {  int n = timePoints.size();  int gap = 1440;  if(n>gap) return 0;  vector<int> minutes(gap);  for(auto tp : timePoints)  {  int t = getTime(tp);  if(minutes[t] > 0) return 0;  minutes[t] = 1;  }  int start = 0;  while(minutes[start] == 0) start++;  int res = gap;  for(int i=start+1; start<gap; ++i)  {  int next = i%gap;  if(minutes[next] != 0)  {  res = min(res, (next - start + gap)%gap);  start = i;  }  }  return res;  }  }; |
| Codility [PermMissingElem](https://codility.com/demo/results/trainingERGKYH-V2S/)  A zero-indexed array A consisting of N different integers is given. The array contains integers in the range [1..(N + 1)], which means that exactly one element is missing.  Your goal is to find that missing element.  Write a function:  int solution(vector<int> &A);  that, given a zero-indexed array A, returns the value of the missing element.  For example, given array A such that:  A[0] = 2 A[1] = 3 A[2] = 1 A[3] = 5  the function should return 4, as it is the missing element.  Assume that:   * N is an integer within the range [0..100,000]; * the elements of A are all distinct; * each element of array A is an integer within the range [1..(N + 1)].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  //empty list return 1  C++1 error from N\*(N+1)/2 may > INT\_MAX  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  n = A.size()+1;  return n\*(n+1)/2 – accumulate(A.begin(), A.end(), 0);  }  C++2  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **long** n = A.size()+1;  n = n\*(n+1)/2;  **for**(**auto** a : A)  n -= a;  **return** n;  }  Testcases  empty\_and\_single  empty list and single element  [▶](https://codility.com/demo/results/trainingV5ZSFJ-4BR/)  missing\_first\_or\_last  the first or the last element is missing  [▶](https://codility.com/demo/results/trainingV5ZSFJ-4BR/)  single  single element  [▶](https://codility.com/demo/results/trainingV5ZSFJ-4BR/)  double  two elements  [▶](https://codility.com/demo/results/trainingV5ZSFJ-4BR/)  simple  simple test  [▶](https://codility.com/demo/results/trainingV5ZSFJ-4BR/)  medium1  medium test, length = ~10,000  [▶](https://codility.com/demo/results/trainingV5ZSFJ-4BR/)  medium2  medium test, length = ~10,000  [▶](https://codility.com/demo/results/trainingV5ZSFJ-4BR/)  large\_range  range sequence, length = ~100,000  [▶](https://codility.com/demo/results/trainingV5ZSFJ-4BR/)  large1  large test, length = ~100,000  [▶](https://codility.com/demo/results/trainingV5ZSFJ-4BR/)  large2  large test, length = ~100,000 |
| Codility [TapeEquilibrium](https://codility.com/demo/results/trainingMRPGTD-U9H/#task-0)  A non-empty zero-indexed array A consisting of N integers is given. Array A represents numbers on a tape.  Any integer P, such that 0 < P < N, splits this tape into two non-empty parts: A[0], A[1], ..., A[P − 1] and A[P], A[P + 1], ..., A[N − 1].  The *difference* between the two parts is the value of: |(A[0] + A[1] + ... + A[P − 1]) − (A[P] + A[P + 1] + ... + A[N − 1])|  In other words, it is the absolute difference between the sum of the first part and the sum of the second part.  For example, consider array A such that:  A[0] = 3 A[1] = 1 A[2] = 2 A[3] = 4 A[4] = 3  We can split this tape in four places:   * P = 1, difference = |3 − 10| = 7 * P = 2, difference = |4 − 9| = 5 * P = 3, difference = |6 − 7| = 1 * P = 4, difference = |10 − 3| = 7   Write a function:  int solution(vector<int> &A);  that, given a non-empty zero-indexed array A of N integers, returns the minimal difference that can be achieved.  For example, given:  A[0] = 3 A[1] = 1 A[2] = 2 A[3] = 4 A[4] = 3  the function should return 1, as explained above.  Assume that:   * N is an integer within the range [2..100,000]; * each element of array A is an integer within the range [−1,000..1,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  #**include** <algorithm>  #**include** <climits>  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **vector**<**int**> B(A.begin(), A.end());  **int** n = A.size()-1;  **for**(**int** i=1; i<n; ++i)  {  A[i] += A[i-1];  B[n-i] += B[n-i+1];  }  **int** res = INT\_MAX;  **for**(**int** i=0; i<n; ++i)  res = min(res, **abs**(A[i]-B[i+1]));  **return** res;  }  Testcases  double  two elements  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  simple\_positive  simple test with positive numbers, length = 5  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  simple\_negative  simple test with negative numbers, length = 5  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  small\_random  random small, length = 100  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  small\_range  range sequence, length = ~1,000  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  small  small elements  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  medium\_random1  random medium, numbers from 0 to 100, length = ~10,000  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  medium\_random2  random medium, numbers from -1,000 to 50, length = ~10,000  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  large\_ones  large sequence, numbers from -1 to 1, length = ~100,000  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  large\_random  random large, length = ~100,000  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  large\_sequence  large sequence, length = ~100,000  [▶](https://codility.com/demo/results/trainingMRPGTD-U9H/)  large\_extreme  large test with maximal and minimal values, length = ~100,000 |
| Codility [MissingInteger](https://codility.com/demo/results/trainingFK89YP-6GH/)  given a non-empty zero-indexed array A of N integers, returns the minimal positive integer (greater than 0) that does not occur in A.  For example, given:  A[0] = 1 A[1] = 3 A[2] = 6 A[3] = 4 A[4] = 1 A[5] = 2  the function should return 5.  Assume that:   * N is an integer within the range [1..100,000]; * each element of array A is an integer within the range [−2,147,483,648..2,147,483,647].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size();  **vector**<**bool**> bucket(n+1);  **for**(**int** a : A)  **if**(a>0 && a<=n)  bucket[a] = **true**;  **for**(**int** i=1; i<=n; ++i)  **if**(bucket[i] == **false**)  **return** i;  **return** n+1;  }  Testcases  extreme\_single  a single element  [▶](https://codility.com/demo/results/trainingFK89YP-6GH/)  simple  simple test  [▶](https://codility.com/demo/results/trainingFK89YP-6GH/)  extreme\_min\_max\_int  MININT and MAXINT (with minus)  [▶](https://codility.com/demo/results/trainingFK89YP-6GH/)  positive\_only  shuffled sequence of 0...100 and then 102...200  [▶](https://codility.com/demo/results/trainingFK89YP-6GH/)  negative\_only  shuffled sequence -100 ... -1  [▶](https://codility.com/demo/results/trainingFK89YP-6GH/)  medium  chaotic sequences length=10005 (with minus)  [▶](https://codility.com/demo/results/trainingFK89YP-6GH/)  large\_1  chaotic + sequence 1, 2, ..., 40000 (without minus)  [▶](https://codility.com/demo/results/trainingFK89YP-6GH/)  large\_2  shuffled sequence 1, 2, ..., 100000 (without minus)  [▶](https://codility.com/demo/results/trainingFK89YP-6GH/)  large\_3  chaotic + many -1, 1, 2, 3 (with minus) |
| Codility  [PermCheck](https://codility.com/demo/results/trainingSGJ754-KND/#task-0)  A non-empty zero-indexed array A consisting of N integers is given.  A *permutation* is a sequence containing each element from 1 to N once, and only once.  For example, array A such that:  A[0] = 4 A[1] = 1 A[2] = 3 A[3] = 2  is a permutation, but array A such that:  A[0] = 4 A[1] = 1 A[2] = 3  is not a permutation, because value 2 is missing.  The goal is to check whether array A is a permutation.  Write a function:  int solution(vector<int> &A);  that, given a zero-indexed array A, returns 1 if array A is a permutation and 0 if it is not.  For example, given array A such that:  A[0] = 4 A[1] = 1 A[2] = 3 A[3] = 2  the function should return 1.  Given array A such that:  A[0] = 4 A[1] = 1 A[2] = 3  the function should return 0.  Assume that:   * N is an integer within the range [1..100,000]; * each element of array A is an integer within the range [1..1,000,000,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size();  **vector**<**bool**> bucket(n+1);  **for**(**int** a : A)  **if**(a>0 && a<=n)  bucket[a] = **true**;  **for**(**int** i=1; i<=n; ++i)  **if**(bucket[i] == **false**)  **return** 0;  **return** 1;  }  extreme\_min\_max  single element with minimal/maximal value  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  single  single element  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  double  two elements  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  antiSum1  total sum is correct, but it is not a permutation, N <= 10  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  small\_permutation  permutation + one element occurs twice, N = ~100  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  medium\_permutation  permutation + few elements occur twice, N = ~10,000  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  antiSum2  total sum is correct, but it is not a permutation, N = ~100,000  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  large\_permutation  permutation + one element occurs three times, N = ~100,000  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  large\_range  sequence 1, 2, ..., N, N = ~100,000  [▶](https://codility.com/demo/results/trainingSGJ754-KND/)  extreme\_values  all the same values, N = ~100,000 |
| Codility  [FrogRiverOne](https://codility.com/demo/results/trainingVY7E5S-3XN/)  A small frog wants to get to the other side of a river. The frog is initially located on one bank of the river (position 0) and wants to get to the opposite bank (position X+1). Leaves fall from a tree onto the surface of the river.  You are given a zero-indexed array A consisting of N integers representing the falling leaves. A[K] represents the position where one leaf falls at time K, measured in seconds.  The goal is to find the earliest time when the frog can jump to the other side of the river. The frog can cross only when leaves appear at every position across the river from 1 to X (that is, we want to find the earliest moment when all the positions from 1 to X are covered by leaves). You may assume that the speed of the current in the river is negligibly small, i.e. the leaves do not change their positions once they fall in the river.  For example, you are given integer X = 5 and array A such that:  A[0] = 1 A[1] = 3 A[2] = 1 A[3] = 4 A[4] = 2 A[5] = 3 A[6] = 5 A[7] = 4  In second 6, a leaf falls into position 5. This is the earliest time when leaves appear in every position across the river.  Write a function:  int solution(int X, vector<int> &A);  that, given a non-empty zero-indexed array A consisting of N integers and integer X, returns the earliest time when the frog can jump to the other side of the river.  If the frog is never able to jump to the other side of the river, the function should return −1.  For example, given X = 5 and array A such that:  A[0] = 1 A[1] = 3 A[2] = 1 A[3] = 4 A[4] = 2 A[5] = 3 A[6] = 5 A[7] = 4  the function should return 6, as explained above.  Assume that:   * N and X are integers within the range [1..100,000]; * each element of array A is an integer within the range [1..X].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(X), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**int** X, **vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **vector**<**bool**> bucket(X+1);  **int** count=0, n = A.size();  **for**(**int** i=0; i<n; ++i)  {  **if**(bucket[A[i]] == **false**)  {  bucket[A[i]] = **true**;  count++;  **if**(count == X)  **return** i;  }  }  **return** -1;  }  Testcases  simple  simple test  [▶](https://codility.com/demo/results/trainingVY7E5S-3XN/)  single  single element  [▶](https://codility.com/demo/results/trainingVY7E5S-3XN/)  extreme\_frog  frog never across the river  [▶](https://codility.com/demo/results/trainingVY7E5S-3XN/)  small\_random1  3 random permutation, X = 50  [▶](https://codility.com/demo/results/trainingVY7E5S-3XN/)  small\_random2  5 random permutation, X = 60  [▶](https://codility.com/demo/results/trainingVY7E5S-3XN/)  extreme\_leaves  all leaves in the same place  [▶](https://codility.com/demo/results/trainingVY7E5S-3XN/)  medium\_random  6 and 2 random permutations, X = ~5,000  [▶](https://codility.com/demo/results/trainingVY7E5S-3XN/)  medium\_range  arithmetic sequences, X = 5,000  [▶](https://codility.com/demo/results/trainingVY7E5S-3XN/)  large\_random  10 and 100 random permutation, X = ~10,000  [▶](https://codility.com/demo/results/trainingVY7E5S-3XN/)  large\_permutation  permutation tests  [▶](https://codility.com/demo/results/trainingVY7E5S-3XN/)  large\_range  arithmetic sequences, X = 30,000 |
| Codility [MaxCounters](https://codility.com/demo/results/training7KHWQ9-8JY/)  You are given N counters, initially set to 0, and you have two possible operations on them:   * *increase(X)* − counter X is increased by 1, * *max counter* − all counters are set to the maximum value of any counter.   A non-empty zero-indexed array A of M integers is given. This array represents consecutive operations:   * if A[K] = X, such that 1 ≤ X ≤ N, then operation K is increase(X), * if A[K] = N + 1 then operation K is max counter.   For example, given integer N = 5 and array A such that:  A[0] = 3 A[1] = 4 A[2] = 4 A[3] = 6 A[4] = 1 A[5] = 4 A[6] = 4  the values of the counters after each consecutive operation will be:  (0, 0, 1, 0, 0) (0, 0, 1, 1, 0) (0, 0, 1, 2, 0) (2, 2, 2, 2, 2) (3, 2, 2, 2, 2) (3, 2, 2, 3, 2) (3, 2, 2, 4, 2)  The goal is to calculate the value of every counter after all operations.  Write a function:  vector<int> solution(int N, vector<int> &A);  that, given an integer N and a non-empty zero-indexed array A consisting of M integers, returns a sequence of integers representing the values of the counters.  The sequence should be returned as:   * a structure Results (in C), or * a vector of integers (in C++), or * a record Results (in Pascal), or * an array of integers (in any other programming language).   For example, given:  A[0] = 3 A[1] = 4 A[2] = 4 A[3] = 6 A[4] = 1 A[5] = 4 A[6] = 4  the function should return [3, 2, 2, 4, 2], as explained above.  Assume that:   * N and M are integers within the range [1..100,000]; * each element of array A is an integer within the range [1..N + 1].   Complexity:   * expected worst-case time complexity is O(N+M); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **vector**<**int**> solution(**int** N, **vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **vector**<**int**> res(N);  **int** preMax=0, curMax=0;  **for**(**int** a : A)  {  **if**(a > N) preMax = curMax;  **else**  {  **if**(res[a-1] < preMax) res[a-1] = preMax+1;  **else** res[a-1]++;    **if**(curMax<res[a-1]) curMax = res[a-1];  }  }  **for**(**int** i=0; i<N; ++i)  **if**(res[i]<preMax)  res[i] = preMax;  **return** res;  }  Testcase  extreme\_small  all max\_counter operations  [▶](https://codility.com/demo/results/training7KHWQ9-8JY/)  single  only one counter  [▶](https://codility.com/demo/results/training7KHWQ9-8JY/)  small\_random1  small random test, 6 max\_counter operations  [▶](https://codility.com/demo/results/training7KHWQ9-8JY/)  small\_random2  small random test, 10 max\_counter operations  [▶](https://codility.com/demo/results/training7KHWQ9-8JY/)  medium\_random1  medium random test, 50 max\_counter operations  [▶](https://codility.com/demo/results/training7KHWQ9-8JY/)  medium\_random2  medium random test, 500 max\_counter operations  [▶](https://codility.com/demo/results/training7KHWQ9-8JY/)  large\_random1  large random test, 2120 max\_counter operations  [▶](https://codility.com/demo/results/training7KHWQ9-8JY/)  large\_random2  large random test, 10000 max\_counter operations  [▶](https://codility.com/demo/results/training7KHWQ9-8JY/)  extreme\_large  all max\_counter operations |
| Codility [FloodDepth](https://codility.com/demo/results/trainingS3W3D5-D9J/)  You are helping a geologist friend investigate an area with mountain lakes. A recent heavy rainfall has flooded these lakes and their water levels have reached the highest possible point. Your friend is interested to know the maximum depth in the deepest part of these lakes.  We simplify the problem in 2-D dimensions. The whole landscape can be divided into small blocks and described by an array A of length N. Each element of A is the altitude of the rock floor of a block (i.e. the height of this block when there is no water at all). After the rainfall, all the low-lying areas (i.e. blocks that have higher blocks on both sides) are holding as much water as possible. You would like to know the maximum depth of water after this entire area is flooded. You can assume that the altitude outside this area is zero and the outside area can accommodate infinite amount of water.  For example, consider array A such that:  A[0] = 1 A[1] = 3 A[2] = 2 A[3] = 1 A[4] = 2 A[5] = 1 A[6] = 5 A[7] = 3 A[8] = 3 A[9] = 4 A[10] = 2  The following picture illustrates the landscape after it has flooded:  https://codility-frontend-prod.s3.amazonaws.com/media/task_static/flood_depth/static/images/auto/567c5fa410e5eec80b633b00c33eb77d.png  The gray area is the rock floor described by the array A above and the blue area with dashed lines represents the water filling the low-lying areas with maximum possible volume. Thus, blocks 3 and 5 have a water depth of 2 while blocks 2, 4, 7 and 8 have a water depth of 1. Therefore, the maximum water depth of this area is 2.  Write a function:  int solution(vector<int> &A);  that, given a non-empty zero-indexed array A consisting of N integers, returns the maximum depth of water.  Given array A shown above, the function should return 2, as explained above.  For the following array:  A[0] = 5 A[1] = 8  the function should return 0, because this landscape cannot hold any water.  Assume that:   * N is an integer within the range [1..100,000]; * each element of array A is an integer within the range [1..100,000,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size(), res = 0;  **vector**<**int**> depths(n);  **int** wall = A[0];  **for**(**int** i=1; i<n-1; ++i)  {  **if**(A[i] > wall) wall = A[i];  **else** depths[i] = wall - A[i];  }    wall = A[n-1];  **for**(**int** i= n-2; i>0; --i)  {  **if**(A[i] > wall)  {  wall = A[i];  depths[i] = 0;  }  **else** depths[i] = min(wall-A[i], depths[i]);    res = max(res, depths[i]);  }  **return** res;  }  Testcase  boundary  max-min values and some random tests, N <= 15  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  triple\_permutation  all permutations of three elements [1, 2, 3]  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  small\_functional  small functional tests, N <= 15  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  small\_random  small random tests, N <= 100  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  multiple\_valleys  many small valleys, N <= 100  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  small\_valley  one small valley, N <= 100  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  medium\_tests  medium tests, N <= 600  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  big\_random  random arrays, N ~= 100,000  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  increasing\_decreasing  randomized values which grow when distance from the center decreases, N ~= 100,000  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  many\_multiple\_valleys  many small valleys, N ~= 100,000  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  big\_valley  one big valley, N ~= 100,000  [▶](https://codility.com/demo/results/trainingS3W3D5-D9J/)  big\_jagged  one big valley with jagged slopes, N ~= 100,000 |
| 11. Container With Most Water  Given *n* non-negative integers *a1*, *a2*, ..., *an*, where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) 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 and *n* is at least 2.  C++1  int maxArea(vector<int>& height) {  int i=0, j=height.size()-1;  int res = 0;  while(i<j)  {  int minH = min(height[i], height[j]);  res = max(res, (j-i)\*minH);  while(height[i]<=minH && i<j) i++;  while(height[j]<=minH && i<j) j--;  }  return res;  } |
| 407. 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.  https://leetcode.com/static/images/problemset/rainwater_empty.png 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.  https://leetcode.com/static/images/problemset/rainwater_fill.png After the rain, water are trapped between the blocks. The total volume of water trapped is 4.  C++1  class Solution {  public:  struct Cell{  int row;  int col;  int height;  Cell(int r, int c, int h) : row(r), col(c), height(h) {}  };  int trapRainWater(vector<vector<int>>& heightMap) {  auto myComp = []( Cell a, Cell b ) { return a.height > b.height; };  priority\_queue<Cell, vector<Cell>, decltype(myComp)> pq(myComp);  int h = heightMap.size();  int w = (h>0) ? heightMap[0].size() : 0;  vector<vector<bool>> visited(h, vector<bool>(w));  for(int j=0; j<w; ++j)  {  visited[0][j] = visited[h-1][j] = true;  pq.push(Cell(0, j, heightMap[0][j]));  pq.push(Cell(h-1, j, heightMap[h-1][j]));  }  for(int i=0; i<h; ++i)  {  visited[i][0] = visited[i][w-1] = true;  pq.push(Cell(i, 0, heightMap[i][0]));  pq.push(Cell(i, w-1, heightMap[i][w-1]));  }    int res = 0;  int edges[4][2] = {{-1,0},{1,0},{0,-1}, {0,1}};  while(!pq.empty())  {  Cell cell = pq.top();  pq.pop();  for(int i=0; i<4; ++i)  {  int row = cell.row + edges[i][0];  int col = cell.col + edges[i][1];  if(row>=0 && row<h && col>=0 && col<w && !visited[row][col])  {  visited[row][col] = true;  res += max(0, cell.height-heightMap[row][col]);  pq.push(Cell(row, col, max(cell.height, heightMap[row][col])));  }  }  }  return res;  }  }; |
| 75. 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.](https://leetcode.com/problems/sort-colors/)  **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?  C++1  void sortColors(vector<int>& nums) {  int i=0, j=0;  for(int k=0; k<nums.size(); ++k)  {  int tem = nums[k];  nums[k] = 2;  if(tem < 2) nums[j++] = 1;  if(tem == 0) nums[i++] = 0;  }  } |
| 73. Set Matrix Zeroes (My SIG interview Q2)  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.](https://leetcode.com/problems/set-matrix-zeroes/)  **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?  C++1  void setZeroes(vector<vector<int>>& matrix) {  int h = matrix.size();  int w = (h>0) ? matrix[0].size() : 0;  vector<bool> row(h);  vector<bool> col(w);  for(int i=0; i<h; ++i)  for(int j=0; j<w; ++j)  if(matrix[i][j] == 0)  row[i]=col[j]=true;  for(int i=0; i<h; ++i)  for(int j=0; j<w; ++j)  if(row[i] || col[j])  matrix[i][j] = 0;  } |
| 517. 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 <-- 5 => 1 1 4  2nd move: 1 <-- 1 <-- 4 => 2 1 3  3rd move: 2 1 <-- 3 => 2 2 2  **Example2**  **Input:** [0,3,0]  **Output:** 2  **Explanation:**  1st move: 0 <-- 3 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 dresses.  **Note:**   1. The range of n is [1, 10000]. 2. The range of dresses number in a super washing machine is [0, 1e5].   C++1  int findMinMoves(vector<int>& machines) {  int sum = accumulate(machines.begin(), machines.end(), 0);  int n = machines.size();  int ave = sum/n;  if(ave\*n != sum) return -1; //if(sum%ave != 0) return -1;  int res = 0, steps=0;  for(int m : machines)  {  steps += m - ave;  res = max(res, max(m-ave, abs(steps)));  }  return res;  } |
| 605. Can Place Flowers  Suppose you have a long flowerbed in which some of the plots are planted and some are not. However, flowers cannot be planted in adjacent plots - they would compete for water and both would die.  Given a flowerbed (represented as an array containing 0 and 1, where 0 means empty and 1 means not empty), and a number **n**, return if **n** new flowers can be planted in it without violating the no-adjacent-flowers rule.  **Example 1:**  **Input:** flowerbed = [1,0,0,0,1], n = 1  **Output:** True  **Example 2:**  **Input:** flowerbed = [1,0,0,0,1], n = 2  **Output:** False  **Note:**   1. The input array won't violate no-adjacent-flowers rule. 2. The input array size is in the range of [1, 20000]. 3. **n** is a non-negative integer which won't exceed the input array size.   C++1  bool canPlaceFlowers(vector<int>& flowerbed, int n) {  for(int i=0; i<flowerbed.size() && n>0; ++i){  if(flowerbed[i]==0)  {  if((i-1==-1 || flowerbed[i-1]==0) && (i+1==flowerbed.size() || flowerbed[i+1]==0))  {  flowerbed[i] = 1;  n--;  }  }  }  return n==0;  } |
|  |

#### Prefix sums

|  |
| --- |
| Codility [CountDiv](https://codility.com/demo/results/trainingZHQUCB-SVS/)  given three integers A, B and K, returns the number of integers within the range [A..B] that are divisible by K, i.e.:  { i : A ≤ i ≤ B, i mod K = 0 }  For example, for A = 6, B = 11 and K = 2, your function should return 3, because there are three numbers divisible by 2 within the range [6..11], namely 6, 8 and 10.  Assume that:   * A and B are integers within the range [0..2,000,000,000]; * K is an integer within the range [1..2,000,000,000]; * A ≤ B.   Complexity:   * expected worst-case time complexity is O(1); * expected worst-case space complexity is O(1).   C++1  **int** **solution**(**int** A, **int** B, **int** K) {  **int** res = B/K - A/K;  **return** (A%K==0) ? res+1 : res;  }  Testcase  simple  A = 11, B = 345, K = 17  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  minimal  A = B in {0,1}, K = 11  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  extreme\_ifempty  A = 10, B = 10, K in {5,7,20}  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  extreme\_endpoints  verify handling of range endpoints, multiple runs  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  big\_values  A = 100, B=123M+, K=2  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  big\_values2  A = 101, B = 123M+, K = 10K  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  big\_values3  A = 0, B = MAXINT, K in {1,MAXINT}  [▶](https://codility.com/demo/results/trainingZHQUCB-SVS/)  big\_values4  A, B, K in {1,MAXINT} |
| Codility [PassingCars](https://codility.com/demo/results/trainingD3QT4A-W7Q/)  A non-empty zero-indexed array A consisting of N integers is given. The consecutive elements of array A represent consecutive cars on a road.  Array A contains only 0s and/or 1s:   * 0 represents a car traveling east, * 1 represents a car traveling west.   The goal is to count passing cars. We say that a pair of cars (P, Q), where 0 ≤ P < Q < N, is passing when P is traveling to the east and Q is traveling to the west.  For example, consider array A such that:  A[0] = 0 A[1] = 1 A[2] = 0 A[3] = 1 A[4] = 1  We have five pairs of passing cars: (0, 1), (0, 3), (0, 4), (2, 3), (2, 4).  Write a function:  int solution(vector<int> &A);  that, given a non-empty zero-indexed array A of N integers, returns the number of pairs of passing cars.  The function should return −1 if the number of pairs of passing cars exceeds 1,000,000,000.  For example, given:  A[0] = 0 A[1] = 1 A[2] = 0 A[3] = 1 A[4] = 1  the function should return 5, as explained above.  Assume that:   * N is an integer within the range [1..100,000]; * each element of array A is an integer that can have one of the following values: 0, 1.   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size()-1;  **for**(**int** i=n, count=0; i>0; --i)  **if**(A[i]==1) A[i] = ++count;  **int** res = 0;  **for**(**int** i=0, num0=1; i<n; ++i)  {  **if**(A[i]==0 && A[i+1]>0)  {  **int** pre = res;  res += num0\*A[i+1];  **if**(res>1000000000 || (res-pre)/num0 != A[i+1]) **return** -1;  }  **else** **if**(A[i]==0) num0++;  **else** num0 = 1;  }  **return** res;  }  C++2  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size()-1;  **for**(**int** i=n, count=0; i>0; --i)  **if**(A[i]==1) A[i] = ++count;  **long** res = 0;  **for**(**int** i=0, num0=1; i<n; ++i)  {  **if**(A[i]==0 && A[i+1]>0)  {  res += (**long**)num0\*A[i+1];  **if**(res>1000000000) **return** -1;  }  **else** **if**(A[i]==0) num0++;  **else** num0 = 1;  }  **return** res;  }  Testcase  single  single element  [▶](https://codility.com/demo/results/training22ZTP7-4YA/)  double  two elements  [▶](https://codility.com/demo/results/training22ZTP7-4YA/)  simple  simple test  [▶](https://codility.com/demo/results/training22ZTP7-4YA/)  small\_random  random, length = 100  [▶](https://codility.com/demo/results/training22ZTP7-4YA/)  small\_random2  random, length = 1000  [▶](https://codility.com/demo/results/training22ZTP7-4YA/)  medium\_random  random, length = ~10,000  [▶](https://codility.com/demo/results/training22ZTP7-4YA/)  large\_random  random, length = ~100,000  1.  large\_big\_answer  0..01..1, length = ~100,000  ✘  WRONG ANSWER  got -1 expected 1000000000  [▶](https://codility.com/demo/results/training22ZTP7-4YA/)  large\_alternate  0101..01, length = ~100,000  ✘  WRONG ANSWER  got -1 expected 999961560  [▶](https://codility.com/demo/results/training22ZTP7-4YA/)  large\_extreme  large test with all 1s/0s, length = ~100,000 |
| Codility  [GenomicRangeQuery](https://codility.com/demo/results/trainingMDHV52-44P/)  A DNA sequence can be represented as a string consisting of the letters A, C, G and T, which correspond to the types of successive nucleotides in the sequence. Each nucleotide has an *impact factor*, which is an integer. Nucleotides of types A, C, G and T have impact factors of 1, 2, 3 and 4, respectively. You are going to answer several queries of the form: What is the minimal impact factor of nucleotides contained in a particular part of the given DNA sequence?  The DNA sequence is given as a non-empty string S = S[0]S[1]...S[N-1] consisting of N characters. There are M queries, which are given in non-empty arrays P and Q, each consisting of M integers. The K-th query (0 ≤ K < M) requires you to find the minimal impact factor of nucleotides contained in the DNA sequence between positions P[K] and Q[K] (inclusive).  For example, consider string S = CAGCCTA and arrays P, Q such that:  P[0] = 2 Q[0] = 4 P[1] = 5 Q[1] = 5 P[2] = 0 Q[2] = 6  The answers to these M = 3 queries are as follows:   * The part of the DNA between positions 2 and 4 contains nucleotides G and C (twice), whose impact factors are 3 and 2 respectively, so the answer is 2. * The part between positions 5 and 5 contains a single nucleotide T, whose impact factor is 4, so the answer is 4. * The part between positions 0 and 6 (the whole string) contains all nucleotides, in particular nucleotide A whose impact factor is 1, so the answer is 1.   Write a function:  vector<int> solution(string &S, vector<int> &P, vector<int> &Q);  that, given a non-empty zero-indexed string S consisting of N characters and two non-empty zero-indexed arrays P and Q consisting of M integers, returns an array consisting of M integers specifying the consecutive answers to all queries.  The sequence should be returned as:   * a Results structure (in C), or * a vector of integers (in C++), or * a Results record (in Pascal), or * an array of integers (in any other programming language).   For example, given the string S = CAGCCTA and arrays P, Q such that:  P[0] = 2 Q[0] = 4 P[1] = 5 Q[1] = 5 P[2] = 0 Q[2] = 6  the function should return the values [2, 4, 1], as explained above.  Assume that:   * N is an integer within the range [1..100,000]; * M is an integer within the range [1..50,000]; * each element of arrays P, Q is an integer within the range [0..N − 1]; * P[K] ≤ Q[K], where 0 ≤ K < M; * string S consists only of upper-case English letters A, C, G, T.   Complexity:   * expected worst-case time complexity is O(N+M); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1 (87%)  **vector**<**int**> solution(**string** &S, **vector**<**int**> &P, **vector**<**int**> &Q) {  // write your code in C++14 (g++ 6.2.0)    **int** m = P.size(), n = S.length();  **vector**<**int**> res(m);  **vector**<**vector**<**int**>> count(n, **vector**<**int**>(4));  **vector**<**int**> dna('T'+1);  dna['C']=1; dna['G']=2; dna['T']=3;  count[0][S[0]]=1;  **for**(**int** i=1; i<n; ++i)  {  **for**(**int** j=0; j<4; ++j)  count[i][j] = count[i-1][j];  count[i][dna[S[i]]]++;  }  **for**(**int** i=0; i<m; ++i)  {  **vector**<**int**> temp(4);  **for**(**int** j=0; j<4; ++j)  {  temp[j] = count[Q[i]][j] - count[P[i]][j];  }  temp[dna[S[P[i]]]]++;  **for**(**int** j=0; j<4; ++j)  **if**(temp[j]>0)  {  res[i] = j+1;  **break**;  }  }  **return** res;  }  C++2 (100%)  **vector**<**int**> solution(**string** &S, **vector**<**int**> &P, **vector**<**int**> &Q) {  // write your code in C++14 (g++ 6.2.0)  **int** m = P.size(), n = S.length();  **vector**<**int**> res(m, 4);  **vector**<**vector**<**int**>> count(n+1, **vector**<**int**>(3));  **for**(**int** i=0; i<n; ++i)  {  count[i+1][0] = count[i][0] + (S[i]=='A' ? 1 : 0);  count[i+1][1] = count[i][1] + (S[i]=='C' ? 1 : 0);  count[i+1][2] = count[i][2] + (S[i]=='G' ? 1 : 0);  }  **for**(**int** i=0; i<m; ++i)  {  **for**(**int** j=0; j<3; ++j)  **if**( count[Q[i]+1][j] - count[P[i]][j] > 0)  {  res[i] = j+1;  **break**;  }  }  **return** res;  }  Java  **public** **int**[] solution(String S, **int**[] P, **int**[] Q) {  // write your code in Java SE 8  **int** n = S.length();  **int** pqLen = P.length;  **int**[] res = **new** **int**[pqLen];  Arrays.fill(res, 4);  **int**[][] count = **new** **int**[n+1][3];  **for**(**int** i=0; i<n; ++i){  count[i+1][0] = count[i][0] + (S.charAt(i)=='A' ? 1 : 0);  count[i+1][1] = count[i][1] + (S.charAt(i)=='C' ? 1 : 0);  count[i+1][2] = count[i][2] + (S.charAt(i)=='G' ? 1 : 0);  }    **for**(**int** i=0; i<pqLen; ++i){  **for**(**int** j=0; j<3; ++j){  **int** numFactor = count[Q[i]+1][j] - count[P[i]][j];  **if**(numFactor > 0){  res[i] = j+1;  **break**;  }  }  }  **return** res;  }  Testcase  extreme\_sinlge  single character string  [▶](https://codility.com/demo/results/trainingMDHV52-44P/)  extreme\_double  double character string  [▶](https://codility.com/demo/results/trainingMDHV52-44P/)  simple  simple tests  [▶](https://codility.com/demo/results/trainingMDHV52-44P/)  small\_length\_string  small length simple string  [▶](https://codility.com/demo/results/trainingMDHV52-44P/)  small\_random  small random string, length = ~300  [▶](https://codility.com/demo/results/trainingMDHV52-44P/)  almost\_all\_same\_letters  GGGGGG..??..GGGGGG..??..GGGGGG  ✘  RUNTIME ERROR  tested program terminated unexpectedly  1.  0.034 s  RUNTIME ERROR, tested program terminated unexpectedly  stderr:  [MONITOR] syscall open was blocked!  \*\*\* Error in `./exec.e.binary': free(): invalid next size (fast): 0x00000000016aed40 \*\*\*  2.  [▶](https://codility.com/demo/results/trainingMDHV52-44P/)  large\_random  large random string, length  [▶](https://codility.com/demo/results/trainingMDHV52-44P/)  extreme\_large  all max ranges |
| Codility [MinAvgTwoSlice](https://codility.com/demo/results/trainingFA5UWJ-GAZ/)  A non-empty zero-indexed array A consisting of N integers is given. A pair of integers (P, Q), such that 0 ≤ P < Q < N, is called a *slice* of array A (notice that the slice contains at least two elements). The *average* of a slice (P, Q) is the sum of A[P] + A[P + 1] + ... + A[Q] divided by the length of the slice. To be precise, the average equals (A[P] + A[P + 1] + ... + A[Q]) / (Q − P + 1).  For example, array A such that:  A[0] = 4 A[1] = 2 A[2] = 2 A[3] = 5 A[4] = 1 A[5] = 5 A[6] = 8  contains the following example slices:   * slice (1, 2), whose average is (2 + 2) / 2 = 2; * slice (3, 4), whose average is (5 + 1) / 2 = 3; * slice (1, 4), whose average is (2 + 2 + 5 + 1) / 4 = 2.5.   The goal is to find the starting position of a slice whose average is minimal.  Write a function:  int solution(vector<int> &A);  that, given a non-empty zero-indexed array A consisting of N integers, returns the starting position of the slice with the minimal average. If there is more than one slice with a minimal average, you should return the smallest starting position of such a slice.  For example, given array A such that:  A[0] = 4 A[1] = 2 A[2] = 2 A[3] = 5 A[4] = 1 A[5] = 5 A[6] = 8  the function should return 1, as explained above.  Assume that:   * N is an integer within the range [2..100,000]; * each element of array A is an integer within the range [−10,000..10,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  (wrong answer)  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n=A.size();  **vector**<**int**> sum2(n);  **for**(**int** i=0; i<n-1; ++i)  {  sum2[i] = A[i]+A[i+1];  }  **int** minA=200001, start=-1;  **for**(**int** i=0; i<n-1; ++i)  {  **if**(sum2[i] < minA)  {  minA = sum2[i];  start = i;  }  }  **return** (sum2[start]==sum2[start+1] && A[start]>A[start+1]) ? start+1 : start;  }  Java  **public** **int** **solution**(**int**[] A) {  // write your code in Java SE 8  **int** n = A.length;  **int**[] accSum = **new** **int**[n+1];  **for**(**int** i=1; i<=n; ++i){  accSum[i] = accSum[i-1] + A[i-1];  }    **double** minAve = Integer.MAX\_VALUE;  **int** res = 0;  **for**(**int** i=0; i<n-1; ++i){  **for**(**int** j=i+2; j<=n; ++j){  **double** ave = (**double**)(accSum[j] - accSum[i]) / (**double**)(j-i);  **if**(ave < minAve){  minAve = ave;  res = i;  }  }  }  **return** res;  }  Improve  **public** **int** **solution**(**int**[] A) {  // write your code in Java SE 8  **int** n = A.length;  **int**[] accSum = **new** **int**[n+1];  **for**(**int** i=1; i<=n; ++i){  accSum[i] = accSum[i-1] + A[i-1];  }    **double** minAve = Integer.MAX\_VALUE;  **int** res = 0;  **for**(**int** i=0; i<n-1; ++i){  **for**(**int** j=i+2; j<=n && j<=i+3; ++j){  **double** ave = (**double**)(accSum[j] - accSum[i]) / (**double**)(j-i);  **if**(ave < minAve){  minAve = ave;  res = i;  }  }  }  **return** res;  }  C++1 (100%)  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n=A.size();  **vector**<**double**> sum2(n);  **for**(**int** i=0; i<n-1; ++i)  {  sum2[i] = (**double**)(A[i]+A[i+1])/2;  **if**(i+2<n) sum2[i] = min(sum2[i], (**double**)(A[i]+A[i+1]+A[i+2])/3);  }  **double** minA=10001, start=-1;  **for**(**int** i=0; i<n-1; ++i)  {  **if**(sum2[i] < minA)  {  minA = sum2[i];  start = i;  }  }  **return** start;  }  C++2 (O(1)space)  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n=A.size();  **float** minA=10001, start=-1;  **for**(**int** i=0; i<n-1; ++i)  {  **float** temp = (**float**)(A[i]+A[i+1])/2;  **if**(i+2<n) temp = min(temp, (**float**)(A[i]+A[i+1]+A[i+2])/3);    **if**(temp < minA)  {  minA = temp;  start = i;  }  }  **return** start;  }  Testcase  double\_quadruple  two or four elements  ✘  WRONG ANSWER  got 1 expected 0  [▶](https://codility.com/demo/results/training8YM89Q-YQ7/)  simple1  simple test, the best slice has length 3  ✘  WRONG ANSWER  got 2 expected 5  [▶](https://codility.com/demo/results/training8YM89Q-YQ7/)  simple2  simple test, the best slice has length 3  ✘  WRONG ANSWER  got 3 expected 2  [▶](https://codility.com/demo/results/training8YM89Q-YQ7/)  small\_random  random, length = 100  [▶](https://codility.com/demo/results/training8YM89Q-YQ7/)  medium\_range  increasing, decreasing (legth = ~100) and small functional  ✘  WRONG ANSWER  got 0 expected 3  [▶](https://codility.com/demo/results/training8YM89Q-YQ7/)  medium\_random  random, N = ~700  [▶](https://codility.com/demo/results/training8YM89Q-YQ7/)  large\_ones  numbers from -1 to 1, N = ~100,000  ✘  WRONG ANSWER  got 0 expected 40002  [▶](https://codility.com/demo/results/training8YM89Q-YQ7/)  large\_random  random, N = ~100,000  [▶](https://codility.com/demo/results/training8YM89Q-YQ7/)  extreme\_values  all maximal values, N = ~100,000  [▶](https://codility.com/demo/results/training8YM89Q-YQ7/)  large\_sequence  many seqeneces, N = ~100,000 |
| Codility [Equi](https://codility.com/demo/results/demoRM86JZ-E78/)  This is a demo task.  A zero-indexed array A consisting of N integers is given. An *equilibrium index* of this array is any integer P such that 0 ≤ P < N and the sum of elements of lower indices is equal to the sum of elements of higher indices, i.e.  A[0] + A[1] + ... + A[P−1] = A[P+1] + ... + A[N−2] + A[N−1].  Sum of zero elements is assumed to be equal to 0. This can happen if P = 0 or if P = N−1.  For example, consider the following array A consisting of N = 8 elements:  A[0] = -1 A[1] = 3 A[2] = -4 A[3] = 5 A[4] = 1 A[5] = -6 A[6] = 2 A[7] = 1  P = 1 is an equilibrium index of this array, because:   * A[0] = −1 = A[2] + A[3] + A[4] + A[5] + A[6] + A[7]   P = 3 is an equilibrium index of this array, because:   * A[0] + A[1] + A[2] = −2 = A[4] + A[5] + A[6] + A[7]   P = 7 is also an equilibrium index, because:   * A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6] = 0   and there are no elements with indices greater than 7.  P = 8 is not an equilibrium index, because it does not fulfill the condition 0 ≤ P < N.  Write a function:  int solution(vector<int> &A);  that, given a zero-indexed array A consisting of N integers, returns any of its equilibrium indices. The function should return −1 if no equilibrium index exists.  For example, given array A shown above, the function may return 1, 3 or 7, as explained above.  Assume that:   * N is an integer within the range [0..100,000]; * each element of array A is an integer within the range [−2,147,483,648..2,147,483,647].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size();  **vector**<**long**> left(n);  **vector**<**long**> right(n);  **for**(**int** i=1; i<n; ++i)  {  left[i] = left[i-1] + A[i-1];  right[n-i-1] = right[n-i] + A[n-i];  }  **for**(**int** i=0; i<n; ++i)  **if**(left[i] == right[i])  **return** i;  **return** -1;  }  Testcase  simple  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  extreme\_large\_numbers  Sequence with extremely large numbers testing arithmetic overflow.  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  extreme\_negative\_numbers  Sequence with extremely large numbers testing arithmetic overflow.  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  overflow\_tests1  arithmetic overflow tests  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  overflow\_tests2  arithmetic overflow tests  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  one\_large  one large number at the end of the sequence  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  sum\_0  sequence with sum=0  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  single\_empty  single number or empty array  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  combinations\_of\_two  multiple runs, all pairs of values: -1, 0 and 1  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  combinations\_of\_three  multiple runs, all triples of values -1, 0 and 1  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  small\_pyramid  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  extreme\_max  Maximal size test  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  large\_long\_sequence\_of\_ones  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  large\_long\_sequence\_of\_minus\_ones  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  medium\_pyramid  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  large\_pyramid  Large performance test, O(n^2) solutions should fail.  [▶](https://codility.com/demo/results/demoRM86JZ-E78/)  huge\_pyramid  Large performance test, O(n^2) solutions should fail. |
| 554. Brick Wall  There is a brick wall in front of you. The wall is rectangular and has several rows of bricks. The bricks have the same height but different width. You want to draw a vertical line from the **top** to the **bottom** and cross the **least** bricks.  The brick wall is represented by a list of rows. Each row is a list of integers representing the width of each brick in this row from left to right.  If your line go through the edge of a brick, then the brick is not considered as crossed. You need to find out how to draw the line to cross the least bricks and return the number of crossed bricks.  **You cannot draw a line just along one of the two vertical edges of the wall, in which case the line will obviously cross no bricks.**  **Example:**  **Input:**  [[1,2,2,1],  [3,1,2],  [1,3,2],  [2,4],  [3,1,2],  [1,3,1,1]]  **Output:** 2  **Explanation:**  https://leetcode.com/static/images/problemset/brick_wall.png  **Note:**   1. The width sum of bricks in different rows are the same and won't exceed INT\_MAX. 2. The number of bricks in each row is in range [1,10,000]. The height of wall is in range [1,10,000]. Total number of bricks of the wall won't exceed 20,000.   C++1  int leastBricks(vector<vector<int>>& wall) {  unordered\_map<int, int> sumMap;  for(auto row : wall)  {  int count =0;  for(int i=0; i<row.size()-1; ++i)  {  count += row[i];  sumMap[count]++;  }  }  int maxSum = 0;  for(auto it=sumMap.begin(); it!=sumMap.end(); ++it)  maxSum = max(maxSum, it->second);  return wall.size()-maxSum;  } |
|  |

#### Count

|  |
| --- |
| Codility [Distinct](https://codility.com/demo/results/training7E8TKW-8AE/)  Write a function  int solution(vector<int> &A);  that, given a zero-indexed array A consisting of N integers, returns the number of distinct values in array A.  Assume that:   * N is an integer within the range [0..100,000]; * each element of array A is an integer within the range [−1,000,000..1,000,000].   For example, given array A consisting of six elements such that:  A[0] = 2 A[1] = 1 A[2] = 1 A[3] = 2 A[4] = 3 A[5] = 1  the function should return 3, because there are 3 distinct values appearing in array A, namely 1, 2 and 3.  Complexity:   * expected worst-case time complexity is O(N\*log(N)); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  //unordered\_set is better  **set**<**int**> **set**;  **for**(**int** a : A)  **set**.insert(a);  **return** **set**.size();  }  Testcase  extreme\_empty  empty sequence  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  extreme\_single  sequence of one element  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  extreme\_two\_elems  sequence of three distinct elements  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  extreme\_one\_value  sequence of 10 equal elements  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  extreme\_negative  sequence of negative elements, length=5  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  extreme\_big\_values  sequence with big values, length=5  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  medium1  chaotic sequence of value sfrom [0..1K], length=100  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  medium2  chaotic sequence of value sfrom [0..1K], length=200  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  medium3  chaotic sequence of values from [0..10], length=200  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  large1  chaotic sequence of values from [0..100K], length=10K  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  large\_random1  chaotic sequence of values from [-1M..1M], length=100K  [▶](https://codility.com/demo/results/training7E8TKW-8AE/)  large\_random2  another chaotic sequence of values from [-1M..1M], length=100K |
| Codility [AbsDistinct](https://codility.com/demo/results/trainingKV68RW-XGT/)  A non-empty zero-indexed array A consisting of N numbers is given. The array is sorted in non-decreasing order. The *absolute distinct count* of this array is the number of distinct absolute values among the elements of the array.  For example, consider array A such that:  A[0] = -5 A[1] = -3 A[2] = -1 A[3] = 0 A[4] = 3 A[5] = 6  The absolute distinct count of this array is 5, because there are 5 distinct absolute values among the elements of this array, namely 0, 1, 3, 5 and 6.  Write a function:  int solution(vector<int> &A);  that, given a non-empty zero-indexed array A consisting of N numbers, returns absolute distinct count of array A.  For example, given array A such that:  A[0] = -5 A[1] = -3 A[2] = -1 A[3] = 0 A[4] = 3 A[5] = 6  the function should return 5, as explained above.  Assume that:   * N is an integer within the range [1..100,000]; * each element of array A is an integer within the range [−2,147,483,648..2,147,483,647]; * array A is sorted in non-decreasing order.   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  #include<climits> //for INT\_MIN  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** INT\_MIN = -2147483648;  **unordered\_set**<**int**> **set**;  **int** extreMin = 0;  **for**(**int** a : A)  {  **if**(a == INT\_MIN) extreMin = 1;  **else** **set**.insert(**abs**(a));  }  **return** **set**.size() + extreMin;  }  Testcase  one\_element  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  two\_elements  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  same\_elements  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  simple  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  simple\_no\_zero  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  simple\_no\_same  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  simple\_no\_negative  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  simple\_no\_positive  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  arith\_overlow  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  medium\_chaotic1  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  medium\_chaotic2  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  long\_sequence\_no\_negative  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  long\_sequence\_no\_positive  [▶](https://codility.com/demo/results/trainingKV68RW-XGT/)  long\_sequence |
| Codility [ArrayInversionCount](https://codility.com/demo/results/trainingU2HM9V-B8Z/)  A zero-indexed array A consisting of N integers is given. An *inversion* is a pair of indexes (P, Q) such that P < Q and A[Q] < A[P].  Write a function:  int solution(vector<int> &A);  that computes the number of inversions in A, or returns −1 if it exceeds 1,000,000,000.  Assume that:   * N is an integer within the range [0..100,000]; * each element of array A is an integer within the range [−2,147,483,648..2,147,483,647].   For example, in the following array:  A[0] = -1 A[1] = 6 A[2] = 3 A[3] = 4 A[4] = 7 A[5] = 4  there are four inversions:  (1,2) (1,3) (1,5) (4,5)  so the function should return 4.  Complexity:   * expected worst-case time complexity is O(N\*log(N)); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1 (63%)  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **if**(A.empty()) **return** 0;  **int** res = 0;  **multiset**<**int**> **set**(A.begin(), A.end());  **for**(**unsigned** **int** i=A.size()-1; i>0; --i)  {  **auto** it = **set**.upper\_bound(A[i]);  res += distance(it, **set**.end());  **if**(res>1e9) **return** -1;  **set**.erase(--it);  }  **return** res;  }  wrong  public int solution(int[] A) {  // write your code in Java SE 8  List<Integer> count = new LinkedList<>();  int res = 0;  for(int a : A)  count.add(a);  Collections.sort(count);  for(int a : A){  count.remove(new Integer(a));  int id = Collections.binarySearch(count, a);  int smallerNum = id>0 ? id-1 : id==0 ? 0 : Math.abs(id+1);  res += smallerNum;  if(res<0 || res > 1\_000\_000\_000) return -1;  }  return res;  } |
| 299. Bulls and Cows  You are playing the following [Bulls and Cows](https://en.wikipedia.org/wiki/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.  C++1  string getHint(string secret, string guess) {  int se[10] = {0};  int gu[10] = {0};  int bull = 0, cow=0;  for(int i=0; i<secret.length(); ++i)  {  if(secret[i]==guess[i]) bull++;  else  {  se[secret[i]-'0']++;  gu[guess[i]-'0']++;  }  }  for(int i=0; i<10; ++i)  cow += min(se[i], gu[i]);  return to\_string(bull)+'A'+to\_string(cow)+'B';  }  C++2  string getHint(string secret, string guess) {  int bc[10] = {0}, a = 0, b = 0;  for(int i=0; i<secret.size(); i++)  {  if(secret[i] == guess[i])  a++;  else  {  int si = secret[i] - '0';  int gi = guess[i] - '0';  if(bc[si]++<0) b++;  if(bc[gi]-->0) b++;  }  }  return to\_string(a)+"A" + to\_string(b) + "B";  } |
|  |

#### Window Slice

|  |
| --- |
| Codility [MaxDoubleSliceSum](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  A non-empty zero-indexed array A consisting of N integers is given.  A triplet (X, Y, Z), such that 0 ≤ X < Y < Z < N, is called a *double slice*.  The *sum* of double slice (X, Y, Z) is the total of A[X + 1] + A[X + 2] + ... + A[Y − 1] + A[Y + 1] + A[Y + 2] + ... + A[Z − 1].  For example, array A such that:  A[0] = 3 A[1] = 2 A[2] = 6 A[3] = -1 A[4] = 4 A[5] = 5 A[6] = -1 A[7] = 2  contains the following example double slices:   * double slice (0, 3, 6), sum is 2 + 6 + 4 + 5 = 17, * double slice (0, 3, 7), sum is 2 + 6 + 4 + 5 − 1 = 16, * double slice (3, 4, 5), sum is 0.   The goal is to find the maximal sum of any double slice.  Write a function:  int solution(vector<int> &A);  that, given a non-empty zero-indexed array A consisting of N integers, returns the maximal sum of any double slice.  For example, given:  A[0] = 3 A[1] = 2 A[2] = 6 A[3] = -1 A[4] = 4 A[5] = 5 A[6] = -1 A[7] = 2  the function should return 17, because no double slice of array A has a sum of greater than 17.  Assume that:   * N is an integer within the range [3..100,000]; * each element of array A is an integer within the range [−10,000..10,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size();  A[0] = A[n-1] = 0;  **vector**<**int**> lSum(n-2);  **vector**<**int**> rSum(n-2);  **for**(**int** i=1; i<n-2; ++i)  {  lSum[i] = max(lSum[i-1]+A[i], 0);  rSum[n-i-3] = max(rSum[n-i-2]+A[n-i-1], 0);  }  **int** res = 0;  **for**(**int** i=0; i<n-2; ++i)  res = max(res, lSum[i]+rSum[i]);  **return** res;  }  Java  **public** **int** **solution**(**int**[] A) {  // write your code in Java SE 8  **int** n = A.length;  **int**[] lSum = **new** **int**[n];  **int**[] rSum = **new** **int**[n];  **for**(**int** i=1; i<n-2; i++) //only left: skip the most two right  lSum[i] = Math.max(lSum[i-1]+A[i], 0);    **for**(**int** i=n-2; i>1; i--) //only right: skip the most two left  rSum[i] = Math.max(rSum[i+1]+A[i], 0);    **int** res = 0;  **for**(**int** i=0; i<n-2; i++){  res = Math.max(res, lSum[i]+rSum[i+2]);  }  **return** res;  }  Testcase  simple3  third simple test  [▶](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  negative  all negative numbers  [▶](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  positive  all positive numbers  [▶](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  extreme\_triplet  three elements  [▶](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  small\_random1  random, numbers form -10\*\*4 to 10\*\*4, length = 70  [▶](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  small\_random2  random, numbers from -30 to 30, length = 300  [▶](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  medium\_range  -1000, ..., 1000  [▶](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  large\_ones  random numbers from -1 to 1, length = ~100,000  [▶](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  large\_random  random, length = ~100,000  [▶](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  extreme\_maximal  all maximal values, length = ~100,000 be careful it could be overflow if values are bigger  [▶](https://codility.com/demo/results/trainingGFPJ7K-QU6/)  large\_sequence  many the same small sequences, length = ~100,000 |
| Codility Max Slice Sum  A non-empty zero-indexed array A consisting of N integers is given. A pair of integers (P, Q), such that 0 ≤ P ≤ Q < N, is called a *slice* of array A. The *sum* of a slice (P, Q) is the total of A[P] + A[P+1] + ... + A[Q].  Write a function:  int solution(vector<int> &A);  that, given an array A consisting of N integers, returns the maximum sum of any slice of A.  For example, given array A such that:  A[0] = 3 A[1] = 2 A[2] = -6 A[3] = 4 A[4] = 0  the function should return 5 because:   * (3, 4) is a slice of A that has sum 4, * (2, 2) is a slice of A that has sum −6, * (0, 1) is a slice of A that has sum 5, * no other slice of A has sum greater than (0, 1).   Assume that:   * N is an integer within the range [1..1,000,000]; * each element of array A is an integer within the range [−1,000,000..1,000,000]; * the result will be an integer within the range [−2,147,483,648..2,147,483,647].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** res = -1000000, sum = 0;  **for**(**int** a : A)  {  sum = max(sum+a, a);  res = max(res, sum);  }  **return** res;  }  Java  **public** **int** **solution**(**int**[] A) {  // write your code in Java SE 8  **int** maxSum = A[0];  **int** cur = 0;  **for**(**int** i=0; i<A.length; ++i){  cur += A[i];  maxSum = Math.max(maxSum, cur);  **if**(cur<0) cur = 0;  }  **return** maxSum;  }  Testcase  one\_element  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  two\_elements  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  three\_elements  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  simple  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  extreme\_minimum  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  fifty\_random  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  neg\_const  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  pos\_const  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  high\_low\_1Kgarbage  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  1Kgarbage\_high\_low  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  growing\_saw  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  blocks  [▶](https://codility.com/demo/results/trainingSHV7CS-62K/)  growing\_negative |
| Codility [CountDistinctSlices](https://codility.com/demo/results/trainingDV2MP4-BMZ/)  An integer M and a non-empty zero-indexed array A consisting of N non-negative integers are given. All integers in array A are less than or equal to M.  A pair of integers (P, Q), such that 0 ≤ P ≤ Q < N, is called a *slice* of array A. The slice consists of the elements A[P], A[P + 1], ..., A[Q]. A *distinct slice* is a slice consisting of only unique numbers. That is, no individual number occurs more than once in the slice.  For example, consider integer M = 6 and array A such that:  A[0] = 3 A[1] = 4 A[2] = 5 A[3] = 5 A[4] = 2  There are exactly nine distinct slices: (0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2), (3, 3), (3, 4) and (4, 4).  The goal is to calculate the number of distinct slices.  Write a function:  int solution(int M, vector<int> &A);  that, given an integer M and a non-empty zero-indexed array A consisting of N integers, returns the number of distinct slices.  If the number of distinct slices is greater than 1,000,000,000, the function should return 1,000,000,000.  For example, given integer M = 6 and array A such that:  A[0] = 3 A[1] = 4 A[2] = 5 A[3] = 5 A[4] = 2  the function should return 9, as explained above.  Assume that:   * N is an integer within the range [1..100,000]; * M is an integer within the range [0..100,000]; * each element of array A is an integer within the range [0..M].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(M), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**int** M, **vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **vector**<**int**> **map**(M+1, -1);  **long** res = 0;  **int** start = 0, n = A.size();  **for**(**unsigned** **int** i=0; i<A.size(); ++i)  {  **if**(**map**[A[i]] >= start)  {  res += (**long**)(i-start)\*(i-start+1)/2;  start = **map**[A[i]]+1;  res -= (**long**)(i-start)\*(i-start+1)/2;  **if**(res > 1e9) **return** 1e9;  }  **map**[A[i]] = i;  }  res += (**long**)(n-start)\*(n-start+1)/2;  **return** (res > 1e9) ? 1e9 : res;  }  Testcase  single  single element  [▶](https://codility.com/demo/results/trainingPRZU3E-9E9/)  double  double elements  [▶](https://codility.com/demo/results/trainingPRZU3E-9E9/)  simple1  first simple test  ✘  WRONG ANSWER  got 19 expected 24  [▶](https://codility.com/demo/results/trainingPRZU3E-9E9/)  simple2  second simple test  ✘  WRONG ANSWER  got 36 expected 37  [▶](https://codility.com/demo/results/trainingPRZU3E-9E9/)  small\_random  small random test, length = 100  ✘  WRONG ANSWER  got 317 expected 335  [▶](https://codility.com/demo/results/trainingPRZU3E-9E9/)  medium\_random  medium random test, length = 500  ✘  WRONG ANSWER  got 9457 expected 13062  [▶](https://codility.com/demo/results/trainingPRZU3E-9E9/)  large  large tests, length = ~100,000  ✘  WRONG ANSWER  got 639982 expected 999955  [▶](https://codility.com/demo/results/trainingPRZU3E-9E9/)  large\_range  large range tests, length = ~100,000  [▶](https://codility.com/demo/results/trainingPRZU3E-9E9/)  large\_random  large random tests, length = ~100,000  ✘  WRONG ANSWER  got 23887511 expected 36424325  [▶](https://codility.com/demo/results/trainingPRZU3E-9E9/)  extreme\_the\_same  all the same elements, length = ~100,000 |
| 480. 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.  C++1 (1472ms) O(n2)  vector<double> medianSlidingWindow(vector<int>& nums, int k) {  vector<double> res;  for(int i=0; i<=nums.size()-k; ++i)  {  vector<int> window(nums.begin()+i, nums.begin()+k+i);  nth\_element(window.begin(), window.begin()+k/2, window.end());  double val = window[k/2];  if(k % 2 ==0)  {  nth\_element(window.begin(), window.begin()+(k-1)/2, window.end());  val = (val + window[(k-1)/2])/2;  }  res.push\_back(val);  }  return res;  }  C++2 (Might be very slow) O(n\*nlog(n))  vector<double> medianSlidingWindow(vector<int>& nums, int k) {  vector<double> res;  vector<int> window(nums.begin(), nums.begin()+(k-1));  for(int i=0; i<=nums.size()-k; ++i)  {  window.push\_back(nums[i]);  sort(window.begin(), window.end());  double val = window[k/2];    if(k % 2 ==0)  {  val = (val + window[(k-1)/2])/2;  }  res.push\_back(val);  auto it = lower\_bound(window.begin(), window.end(), nums[i-k+1]);  window.erase(it);  }  return res;  }  //lower\_bound: find the first mached value, no sorting  C++3 (93ms) O(n\*log(n))  vector<double> medianSlidingWindow(vector<int>& nums, int k) {  vector<double> medians;  multiset<int> window(nums.begin(), nums.begin()+k);  auto mid = next(window.begin(), k/2);  for(int i=k; ; ++i)  {  medians.push\_back(((double)\*mid + \*prev(mid, 1-k%2)) / 2);  if(i==nums.size()) return medians;  window.insert(nums[i]);  if(nums[i] < \*mid) mid--;  if(nums[i-k] <= \*mid) mid++;  window.erase(window.lower\_bound(nums[i-k]));  }  return medians;  } |
| 209. Minimum Size Subarray Sum  Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a **contiguous** 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.](https://leetcode.com/problems/minimum-size-subarray-sum/)  **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*).  C++1 O(n\*log(n))  int minSubArrayLen(int s, vector<int>& nums) {  int n = nums.size();  if(n==0 || s==0) return 0;  int res = n+1;  int sum = 0;  for(int i=0; i<n; ++i)  sum[i+1] = sum[i]+nums[i];  for(int i=0; i<n; ++i)  {  int left = i+1;  int right = n;  while(left <= right)  {  int mid = left + (right-left)/2;  if(sum[mid]-sum[i] >= s)  {  res = min(res, mid-i);  right = mid-1;  }  else  left = mid+1;  }  }  return res==n+1 ? 0 : res;  }  C++2 O(n)  int minSubArrayLen(int s, vector<int>& nums) {  int sum =0, start=0, minLen = INT\_MAX;  for(int i=0; i<nums.size(); ++i)  {  sum+=nums[i];  while(sum>=s)  {  minLen = min(minLen, i-start+1);  sum -= nums[start++];  }  }  return minLen==INT\_MAX ? 0:minLen;  }  Java1 O(n\*log(n))  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;  }  //check if has sum with size >= s in array nums  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;  }  } |
| 152. 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.  C++1  int maxProduct(vector<int>& nums) {  int left=1, right=1, res=INT\_MIN, n=nums.size();  for(int i=0; i<n; ++i)  {  left \*= nums[i];  right \*= nums[n-i-1];  res = max(res, max(left, right));  left = (left==0) ? 1 : left;  right = (right==0) ? 1 : right;  }  return res;  } |
| 55. 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.  C++1 (TLE)  class Solution {  public:  bool canReachGoal(vector<int>& nums, vector<bool>& visited, int start, int& end)  {  if(start+nums[start] >=end-1) return true;  visited[start] = true;  bool res = false;  for(int i=start+nums[start]; i>start; --i)  if(visited[i]==false)  res |= canReachGoal(nums, visited, i, end);  return res;  }  bool canJump(vector<int>& nums) {  int n = nums.size();  vector<bool> visited(n);  return canReachGoal(nums, visited, 0, n);  }  };  C++2  bool canJump(vector<int>& nums) {  int start = 0, maxJump = nums[0], len = nums.size();  for(int i=1; i<len && i<=start+maxJump; ++i)  {  if(i-start+nums[i]>maxJump)  {  if(i+nums[i]>=len-1) return true;  start = i;  maxJump = nums[i];  }  }  return start+maxJump>=len-1;  } |
| 134. 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.  C++1  int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {  int start=0, tank=0, mostLackGas = INT\_MAX, n = gas.size();  for(int i=0; i<n; ++i)  {  tank += gas[i]-cost[i];  if(tank<mostLackGas)  {  mostLackGas = tank;  start = i+1; //pass the most lack gas road  }  }  return tank<0 ? -1 : (start%n);  } |
| 3. Longest Substring Without Repeating Characters  Given a string, find the length of the **longest substring** without repeating characters (not only letters)!!!!!.  **Examples:**  Given "abcabcbb", the answer is "abc", which the length is 3.  Given "bbbbb", the answer is "b", with the length of 1.  Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a **substring**, "pwke" is a *subsequence* and not a substring.  C++1  int lengthOfLongestSubstring(string s) {  vector<int> letter(128, -1);  int res = 0;  for(int i=0,j=0; j<s.length(); ++j)  {  if(letter[s[j]] > -1)  {  for(int k=i; k<letter[s[j]]; ++k)  letter[s[k]] = -1;  i = letter[s[j]]+1;  }  res = max(res, j-i+1);  letter[s[j]] = j;  }  return res;  }  C++2  int lengthOfLongestSubstring(string s) {  int maxS = 0, start=-1;  vector<int> dict(256, -1);  for(int i=0; i<s.size(); ++i)  {  if(dict[s[i]]>start) start = dict[s[i]];  dict[s[i]] = i;  maxS = max(maxS, i-start);  }  return maxS;  } |
| 220. 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 **absolute** difference between **nums[i]** and **nums[j]** is at most *t* and the **absolute** difference between *i* and *j* is at most *k*.  C++1 (TLE) bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t) {  set<int> window;  for(int i=0; i<nums.size(); ++i)  {  if(i>k) window.erase(nums[i-k-1]);  auto it = window.lower\_bound(nums[i]-t);  if(it!=window.end() && (\*it - nums[i]) <=t) return true;  window.insert(nums[i]);  }  return false;  }  class Solution {  public:  bool canFind(vector<int>& nums, int k, int t, int start, int end)  {  if(start>=end) return false;  int mid = start + (end-start)/2;  for(int right=mid+1; right<=end && right<=mid+k; ++right)  for(int left=mid; left>=0 && right-left<=k; --left)  if(abs((long)nums[left]-nums[right]) <= t)  return true;  return canFind(nums, k, t, start, mid) || canFind(nums, k, t, mid+1, end);  }  bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t) {  return canFind(nums, k, t, 0, nums.size()-1);  }  };  C++2  bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t) {  set<int> window;  for(int i=0; i<nums.size(); ++i)  {  if(i>k) window.erase(nums[i-k-1]);  auto it = window.lower\_bound(nums[i]-t);  if(it!=window.end() && (\*it - nums[i]) <=t) return true;  window.insert(nums[i]);  }  return false;  } |
|  |

#### Interval

|  |
| --- |
| 228. 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"].  C++1  vector<string> summaryRanges(vector<int>& nums) {  vector<string> res;  int n = nums.size();  for(int i=0; i<n; ++i)  {  if(i+1<n && nums[i]+1==nums[i+1])  {  res.push\_back(to\_string(nums[i]) + "->");  while(i+1<n && nums[i]+1==nums[i+1]) i++;  res.back() += to\_string(nums[i]);  }  else  res.push\_back(to\_string(nums[i]));  }  return res;  }  C++2  vector<string> summaryRanges(vector<int>& nums) {  vector<string> res;  if(nums.empty()) return res;  int i=0, j=0, len=nums.size();  while(j<len)  {  while(j+1<len && nums[j]==nums[j+1]-1) j++;  string s = (i==j) ? to\_string(nums[i]) : (to\_string(nums[i]) + "->" + to\_string(nums[j]));  res.push\_back(s);  i=j=j+1;  }  return res;  } |
| 624. Maximum Distance in Arrays  Given m arrays, and each array is sorted in ascending order. Now you can pick up two integers from two different arrays (each array picks one) and calculate the distance. We define the distance between two integers a and b to be their absolute difference |a-b|. Your task is to find the maximum distance.  **Example 1:**  **Input:**  [[1,2,3],  [4,5],  [1,2,3]]  **Output:** 4  **Explanation:**  One way to reach the maximum distance 4 is to pick 1 in the first or third array and pick 5 in the second array.  **Note:**   1. Each given array will have at least 1 number. There will be at least two non-empty arrays. 2. The total number of the integers in **all** the m arrays will be in the range of [2, 10000]. 3. The integers in the m arrays will be in the range of [-10000, 10000].   C++1  int maxDistance(vector<vector<int>>& arrays) {  int minV=10001, maxV=-10001;  int minId, maxId;  int disMin=0, disMax=0;  for(int i=0; i<arrays.size(); ++i)  {  if(arrays[i][0]<minV)  {  minV = arrays[i][0];  minId = i;  }  if(arrays[i].back() > maxV)  {  maxV = arrays[i].back();  maxId = i;  }  }  for(int i=0; i<arrays.size(); ++i)  {  if(i!=minId)  disMin = max(disMin, arrays[i].back()-minV);  if(i!=maxId)  disMax = max(disMax, maxV-arrays[i][0]);  }  return max(disMin, disMax);  } |
| 621. Task Scheduler  Given a char array representing tasks CPU need to do. It contains capital letters A to Z where different letters represent different tasks.Tasks could be done without original order. Each task could be done in one interval. For each interval, CPU could finish one task or just be idle.  However, there is a non-negative cooling interval **n** that means between two **same tasks**, there must be at least n intervals that CPU are doing different tasks or just be idle.  You need to return the **least** number of intervals the CPU will take to finish all the given tasks.  **Example 1:**  **Input:** tasks = ['A','A','A','B','B','B'], n = 2  **Output:** 8  **Explanation:** A -> B -> idle -> A -> B -> idle -> A -> B.  **Note:**   1. The number of tasks is in the range [1, 10000]. 2. The integer n is in the range [0, 100].   C++1  int leastInterval(vector<char>& tasks, int n) {  vector<int> letters(26);  for(char c : tasks)  letters[c-'A']++;  sort(letters.begin(), letters.end(), greater<int>());  int idle = letters[0]-1;  int idleSum = idle\*n;  for(int i= 1; i<26 && letters[i]>0; ++i)  idleSum -= min(letters[i], idle);  return tasks.size() + ((idleSum>0) ? idleSum : 0);  }  Java1  public int leastInterval(char[] tasks, int n) {  int[] map = new int[26];  for(char c : tasks)  map[c-'A']++;  Arrays.sort(map);  int idle = map[25]-1;  int idleSum = idle\*n;  for(int i=24; i>=0 && map[i]>0; --i)  idleSum -= Math.min(map[i], idle);  return tasks.length + ((idleSum > 0) ? idleSum : 0);  } |
|  |

### 串(String)

|  |
| --- |
| 345. 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".  **Note:** The vowels does not include the letter "y".  C++1 (12ms)  string reverseVowels(string s) {  int vowels[128] = {0};  vowels['a']=vowels['e']=vowels['i']=vowels['o']=vowels['u']= 1;  vowels['A']=vowels['E']=vowels['I']=vowels['O']=vowels['U']= 1;  int i=0, j=s.length()-1;  while(i<j)  {  if(i<j && vowels[s[i]]!=1)  i++;  else if(i<j && vowels[s[j]]!=1)  j--;  else  {  swap(s[i], s[j]);  i++;  j--;  }  }  return s;  }  C++2 (12ms)  bool isVowel(char);  string reverseVowels(string s) {  int i=0, j= s.length()-1;  string v = "aeiouAEIOU";  while(i<j)  {  while(v.find(s[i]) == string::npos && i<j) i++;  while(v.find(s[j]) == string::npos && i<j) j--;  swap(s[i++], s[j--]);  }  return s;  }  Java1 (6ms)  private boolean isVowel(char c)  {  c = Character.toLowerCase(c);  if(c=='a' || c=='e' || c=='i' || c=='o' || c=='u') return true;  return false;  }  public String reverseVowels(String s) {  char[] cs = s.toCharArray();  int i=0, j=s.length()-1;  while(i<j)  {  if(isVowel(cs[i]) && isVowel(cs[j]))  {  char temp = cs[i];  cs[i++] = cs[j];  cs[j--] = temp;  }  else if(isVowel(cs[i])) j--;  else if(isVowel(cs[j])) i++;  else {i++; j--;}  }  return new String(cs);  } |
| 434. Number of Segments in a String  Count the number of segments in a string, where a segment is defined to be a contiguous sequence of non-space characters.  Please note that the string does not contain any **non-printable** characters.  C++1 (3ms)  int countSegments(string s) {  s += " ";  int count=0;  for(int i=1; i<s.length(); ++i)  if(s[i] == ' ' && s[i-1]!=' ') // if(isspace(s[i]) && !isspace(s[i-1]))  count++;  return count;  }  C++2 (0ms)  int countSegments(string s) {  int count=0;  for(int i=0; i<s.size(); ++i)  {  if(!isspace(s[i]))  {  count++;  while(i<s.size() && !isspace(s[i]))  i++;  }  }  return count;  } |
| 38. 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 *n*th sequence.  Note: The sequence of integers will be represented as a string.  C++1 (3ms) O(n2)  string countAndSay(int n) {  string res = "1";  for(int i=2; i<=n; ++i)  {  string aRow="";  int count = 1;  for(int j=1; j<res.size(); ++j)  {  if(res[j]== res[j-1]) count++;  else  {  aRow+=to\_string(count)+res[j-1];  count=1;  }  }  aRow += to\_string(count)+res[res.length()-1];  res = aRow;  }  return res;  }  C++2 (4ms) O(n2)  string countAndSay(int n) {  string s = "1", sub;  for(int i=1; i<n; i++)  {  int count =1, len = s.size() - 1 ;  for(int j=0; j<len; j++)  {  if(s[j] == s[j+1])  {  count++;  }  else  {  sub += to\_string(count) + s[j];  count = 1;  }  }  sub += to\_string(count) + s[len];  s = sub;  sub="";  }  return s;  }  Java1 (6ms) O(n2)  public String countAndSay(int n) {  String s = "1";  for(int i=1; i<n; ++i)  {  StringBuilder sb = new StringBuilder();  for(int j=1, count=1; j<=s.length(); ++j)  {  if(j==s.length() || s.charAt(j-1)!=s.charAt(j))  {  sb.append(count);  sb.append(s.charAt(j-1));  count=1;  }  else  count++;  }  s = sb.toString();  }  return s;  } |
|  |

#### Path

|  |
| --- |
| 71. 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.](https://leetcode.com/problems/simplify-path/)  **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".   C++1  string simplifyPath(string path) {  string res, token;  vector<string> temp;  stringstream ss(path);  while(getline(ss, token, '/'))  {  if(token=="." || token=="") continue;  else if(token=="..")  {  if(!temp.empty()) temp.pop\_back();  }  else temp.push\_back(token);  }  for(string str : temp)  res += "/" + str;  return res.empty() ? "/" : res;  } |
| 609. Find Duplicate File in System  Given a list of directory info including directory path, and all the files with contents in this directory, you need to find out all the groups of duplicate files in the file system in terms of their paths.  A group of duplicate files consists of at least **two** files that have exactly the same content.  A single directory info string in the **input** list has the following format:  "root/d1/d2/.../dm f1.txt(f1\_content) f2.txt(f2\_content) ... fn.txt(fn\_content)"  It means there are **n** files (f1.txt, f2.txt ... fn.txt with content f1\_content, f2\_content ... fn\_content, respectively) in directory root/d1/d2/.../dm. Note that n >= 1 and m >= 0. If m = 0, it means the directory is just the root directory.  The **output** is a list of group of duplicate file paths. For each group, it contains all the file paths of the files that have the same content. A file path is a string that has the following format:  "directory\_path/file\_name.txt"  **Example 1:**  **Input:**  ["root/a 1.txt(abcd) 2.txt(efgh)", "root/c 3.txt(abcd)", "root/c/d 4.txt(efgh)", "root 4.txt(efgh)"]  **Output:**  [["root/a/2.txt","root/c/d/4.txt","root/4.txt"],["root/a/1.txt","root/c/3.txt"]]  **Note:**   1. No order is required for the final output. 2. You may assume the directory name, file name and file content only has letters and digits, and the length of file content is in the range of [1,50]. 3. The number of files given is in the range of [1,20000]. 4. You may assume no files or directories share the same name in a same directory. 5. You may assume each given directory info represents a unique directory. Directory path and file infos are separated by a single blank space.   **Follow up beyond contest:**   1. Imagine you are given a real file system, how will you search files? DFS or BFS ? 2. If the file content is very large (GB level), how will you modify your solution? 3. If you can only read the file by 1kb each time, how will you modify your solution? 4. What is the time complexity of your modified solution? What is the most time consuming part and memory consuming part of it? How to optimize? 5. How to make sure the duplicated files you find are not false positive?   C++1  class Solution {  public:  vector<string> getPathInfo(string path)  {  stringstream ss(path);  string token;  vector<string> res;  while(getline(ss, token, ' '))  res.push\_back(token);  return res;  }    vector<string> getFileInfo(string file)  {  vector<string> res(2);  int i = 0;  while(file[i] != '(') i++;  res[0] = file.substr(0, i);  res[1] = file.substr(i, file.length()-1);  return res;  }  vector<vector<string>> findDuplicate(vector<string>& paths) {  vector<vector<string>> res;  unordered\_map<string, vector<string>> map;  for(string path : paths)  {  vector<string> p = getPathInfo(path);  string dir = p[0]+"/";  for(int i=1; i<p.size(); ++i)  {  vector<string> file = getFileInfo(p[i]);  map[file[1]].push\_back(dir + file[0]);  }  }  for(auto it=map.begin(); it!=map.end(); ++it)  if(it->second.size() > 1)  res.push\_back(it->second);  return res;  }  }; |
|  |

#### Anagrams

|  |
| --- |
| 242. 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?  C++1 (13ms)  bool isAnagram(string s, string t) {  int sLen = s.length();  if(sLen != t.length()) return false;  int count[128] = {0};  for(int i=0; i<sLen; ++i)  count[s[i]]++;  for(char c : t)  if(--count[c] < 0) return false;  return true;  }  C++2 (76ms)  bool isAnagram(string s, string t) {  int lens = s.length();  int lent = t.length();  if(lens != lent) return false;  sort(s.begin(), s.end());  sort(t.begin(), t.end());  for(int i=0; i<lens; i++)  if(s[i] != t[i])  return false;  return true;  }  Java1 (3ms)  public boolean isAnagram(String s, String t) {  int n = s.length();  if(n != t.length()) return false;  int[] alp = new int[26];  for(int i=0; i<n; ++i)  {  alp[s.charAt(i)-'a']++;  alp[t.charAt(i)-'a']--;  }  for(int a : alp)  {  if(a!=0) return false;  }  return true;  } |
| 438. 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".  C++1 (**1375 ms)**  class Solution {  public:  bool isAnagrams(string x, string y)  {  int count[128] = {0};  for(int i=0; i<x.length(); ++i)  {  ++count[x[i]];  --count[y[i]];  }  for(int i=0; i<x.length(); ++i)  {  if(count[x[i]]!=0) return false;  }  return true;  }  vector<int> findAnagrams(string s, string p) {  vector<int> res;  int sLen=s.length(), pLen=p.length();  if(sLen<pLen) return res;  for(int i=0; i<=sLen-pLen; ++i)  {  if(isAnagrams(s.substr(i, pLen), p))  {  res.push\_back(i);  }  }  return res;  }  };  C++2 (**56 ms)**  vector<int> findAnagrams(string s, string p) {  vector<int> res;  if(s.length()==0) return res;  int left=0, right=0, pLen = p.length(), count = pLen;  int letters[128] = {0};  for(char c : p) letters[c]++;  while(right < s.length())  {  if(letters[s[right]]-- > 0)  count--;  right++;    if(count==0) res.push\_back(left);    if(right-left == pLen)  {  if(letters[s[left]]++ >= 0)  count++;  left++;  }  }  return res;  } |
| 49. 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:** All inputs will be in lower-case.  C++1  vector<vector<string>> groupAnagrams(vector<string>& strs) {  vector<vector<string>> res;  unordered\_map<string, int> map;  for(string str : strs)  {  string subStr = str;  sort(subStr.begin(),subStr.end());  if(map.count(subStr)==0)  {  res.push\_back(vector<string>(1, str));  map[subStr] = res.size()-1;  }  else  res[map[subStr]].push\_back(str);  }  return res;  }  C++2  vector<vector<string>> groupAnagrams(vector<string>& strs) {  vector<vector<string>> res;  if(strs.empty()) return res;  string s;  unordered\_map<string, multiset<string>> groups;  for(string str : strs)  {  s = str;  sort(s.begin(), s.end());  groups[s].insert(str);  }  for(auto g : groups)  {  vector<string> arow(g.second.begin(), g.second.end());  res.push\_back(arow);  }  return res;  } |
| 567. Permutation in String  Given two strings **s1** and **s2**, write a function to return true if **s2** contains the permutation of **s1**. In other words, one of the first string's permutations is the **substring** of the second string.  **Example 1:**  **Input:**s1 = "ab" s2 = "eidbaooo"  **Output:**True  **Explanation:** s2 contains one permutation of s1 ("ba").  **Example 2:**  **Input:**s1= "ab" s2 = "eidboaoo"  **Output:** False  **Note:**   1. The input strings only contain lower case letters. 2. The length of both given strings is in range [1, 10,000].   C++1  class Solution {  public:  bool hasSameLetters(string& sub, vector<int>& ls)  {  vector<int> ls2(26,0);  for(char c : sub)  ls2[c-'a']++;  for(int i=0; i<26; ++i)  if(ls[i] != ls2[i]) return false;  return true;  }  bool checkInclusion(string s1, string s2) {  vector<int> ls(26,0);  for(char c : s1)  ls[c-'a']++;  int subS= s1.length();  for(int i=0; i+subS<=s2.size(); ++i)  {  string sub = s2.substr(i, subS);  if(hasSameLetters(sub, ls)) return true;  }  return false;  }  }; |
| 472. 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","ratcatdogcat"]  **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.   C++1 (MLE)  class Solution {  public:  bool isConcatenatedWord(unordered\_map<string, bool>& map, string& word)  {  if(map[word]==true) return true;  for(int i=1; i<word.length(); ++i)  {  string left = word.substr(0, i);  string right = word.substr(i);  if(map[left] == true && isConcatenatedWord(map, right))  {  //map[word] = true;  return true;  }  }  return false;  }  vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {  unordered\_map<string, bool> map;  vector<string> res;  sort(words.begin(), words.end(), [](string a, string b) {return a.length()<b.length();});  map[""] = true;  for(string word : words)  {  if(isConcatenatedWord(map, word))  res.push\_back(word);  map[word] = true;  }  return res;  }  };  C++2 (TLE)  vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {  unordered\_map<string, bool> map;  vector<string> res;  sort(words.begin(), words.end(), [](string a, string b) {return a.length()<b.length();});  int min = words[0].size();  for(string word : words)  {  string temp = word;  for(int i=min; i<temp.length(); ++i)  {  string left = word.substr(0, i);  if(map[left] == true)  {  string right = word.substr(i);  if(map[right]==true) res.push\_back(word);  temp=right;  i=min;  }  }  map[word] = true;  }  return res;  }  C++3 (TLE)  class Solution {  public:  bool isConcatenatedWord(unordered\_map<string, bool>& map, string& word)  {  if(word.empty()) return true;  vector<bool> dp(word.length()+1);  dp[0] = true;  for(int i=1; i<=word.length(); ++i)  {  for(int j=0;j<i; ++j)  {  if(dp[j]==true && map[word.substr(j, i-j)]==true) //left is concatenate, right is in map  {  dp[i] = true;  break;  }  }  }  return dp[word.length()];  }  vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {  unordered\_map<string, bool> map;  vector<string> res;  sort(words.begin(), words.end(), [](string a, string b) {return a.length()<b.length();});  for(string word : words)  {  if(isConcatenatedWord(map, word))  res.push\_back(word);  map[word] = true;  }  return res;  }  };  C++4 (662ms)  vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {  unordered\_map<string, bool> map;  vector<string> res;  sort(words.begin(), words.end(), [](string a, string b) {return a.length()<b.length();});  for(string word : words)  {  int n=word.length();  if(n==0) continue;  vector<int> dp(n+1);  dp[0] = 1;  for(int i=1; i<=n; ++i)  {  for(int j=0;j<i; ++j)  {  if(dp[j]==1 && map[word.substr(j, i-j)]==true) //left is concatenate, right is in map  {  dp[i] = 1;  break;  }  }  }  if(dp[n]==1) res.push\_back(word);  map[word] = true;  }  return res;  }  C++5 (MLE)  class Trie{  public:  vector<Trie\*> child;  bool isWord;  Trie(){  child = vector<Trie\*>(26);  isWord=false; //strange, default should be false, but it seems not  }  };  class Solution {  public:  void addWord(string& word, Trie\* root)  {  Trie\* cur = root;  for(char w : word)  {  if(cur->child[w-'a']==NULL)  cur->child[w-'a'] = new Trie();  cur = cur->child[w-'a'];  }  cur->isWord = true;  }  bool isConcatenateWord(string& word, int start, Trie\* root, int wordCount)  {  Trie\* cur = root;  int n = word.length();  for(int i=start; i<n; ++i)  {  if(cur->child[word[i]-'a']==NULL) return false;    if(cur->child[word[i]-'a']->isWord == true)  {  if(i==n-1) return wordCount>=1;  if(isConcatenateWord(word, i+1, root, wordCount+1))  return true;  }  cur = cur->child[word[i]-'a'];  }  return false;  }  vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {  Trie\* root = new Trie();  vector<string> res;  for(int i=0; i<words.size(); ++i)  {  if(!words[i].empty())  addWord(words[i], root);  }  for(int i=0; i<words.size(); ++i)  {  if(!words[i].empty() && isConcatenateWord(words[i], 0, root, 0))  res.push\_back(words[i]);  }  return res;  }  }; |
|  |

#### Duplicate

|  |
| --- |
| 187. 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"].  C++1  vector<string> findRepeatedDnaSequences(string s) {  unordered\_map<string, int> map;  vector<string> res;  int n = s.length()-10;  for(int i=0; i<=n; ++i) //for(int i=0; i<=s.length()-10; ++i) has error, don’t know why!  {  string sub = s.substr(i, 10);  if(map[sub]==1) res.push\_back(sub);  map[sub]++;  }  return res;  } |
|  |
|  |

#### Matching

|  |
| --- |
| 205. 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.  C++1 (6ms)  bool isIsomorphic(string s, string t) {  int Scount[128] = {0}, Tcount[128] = {0};  if(s.length()!=t.length()) return false;  for(int i=0; i<s.length(); ++i)  {  if(Scount[s[i]] != Tcount[t[i]]) return false;  if(Scount[s[i]] == 0)  Scount[s[i]] = Tcount[t[i]] = i+1;  }  return true;  }  Java1 (5ms)  public boolean isIsomorphic(String s, String t) {  char[] ss = s.toCharArray();  char[] tt = t.toCharArray();  int[] sAddr = new int[128];  int[] tAddr = new int[128];  for(int i=0; i<s.length(); ++i)  {  char si = ss[i], ti = tt[i];  if(sAddr[si] != tAddr[ti]) return false;  else if(sAddr[si] == 0)  {  sAddr[si] = i+1;  tAddr[ti] = i+1;  }  }  return true;  }  Java2 (15ms)  public boolean isIsomorphic(String s, String t) {  int[] sAddr = new int[256];  int[] tAddr = new int[256];  for(int i=0; i<s.length(); ++i)  {  if(sAddr[s.charAt(i)] != tAddr[t.charAt(i)]) return false;  else  {  sAddr[s.charAt(i)] = i+1;  tAddr[t.charAt(i)] = i+1;  }  }  return true;  } |
| 20. 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.  Java  **public** **int** **solution**(String S) {  // write your code in Java SE 8  **if**(S.length() % 2 != 0) **return** 0;  Deque<Character> stack = **new** LinkedList<>();  **for**(**char** c : S.toCharArray()){  **if**(c==')' || c==']' || c=='}'){  **if**(stack.isEmpty()) **return** 0;  **char** open = stack.pop();  **if**(c==')' && open!='('  || c==']' && open!='['  || c=='}' && open!='{')  **return** 0;  }  **else** stack.push(c);  }  **return** stack.isEmpty() ? 1 : 0;  }  C++ (9ms) O(n)  bool isValid(string s) {  if(s.length()%2 != 0) return false; //3ms  int diff1 = ')'-'(';  int diff2 = ']'-'[';  int diff3 = '}'-'{';  stack<char> brackets;  for(char c : s)  {  if(c==')' || c==']' || c=='}')  {  if(brackets.empty()) return false;  char diff = c - brackets.top();  brackets.pop();  if(diff!= diff1 && diff!=diff2 && diff!=diff3) return false;  }  else  brackets.push(c);  }  return brackets.empty();  }  C++2 (0ms)  bool isValid(string s) {  stack<char> store;  for(auto c : s)  {  if(c=='(' || c=='[' || c=='{')  store.push(c);  else if(!store.empty() && (c-store.top() == 1 || c-store.top() == 2))  store.pop();  else  return false;  }  return store.empty();  }  Java1 (1ms)  public boolean isValid(String s) {  char[] ss = s.toCharArray();  Stack<Character> stack = new Stack<Character>();  for(char c : ss)  {  if(c=='(' || c=='[' || c=='{')  stack.push(c);  else if(!stack.empty() && (c-stack.peek() == 1 || c-stack.peek() == 2))  stack.pop();  else  return false;  }  return stack.empty();  }  Java2 (1ms)  public boolean isValid(String s) {  char[] ss = s.toCharArray();  Stack<Character> stack = new Stack<Character>();  for(char c : ss)  {  if(c=='(' || c=='[' || c=='{')  stack.push(c);  else if(!stack.empty())  {  char top = stack.pop();  int valid = c - top;  if(valid!=1 && valid!=2)  return false;  }  else  return false;  }  return stack.empty();  }  Testcase  negative\_match  invalid structures  empty  empty string  simple\_grouped  simple grouped positive and negative test, length=22  large1  simple large positive test, 100K ('s followed by 100K )'s + )(  large2  simple large negative test, 10K+1 ('s followed by 10K )'s + )( + ()  large\_full\_ternary\_tree  tree of the form T=(TTT) and depth 11, length=177K+  multiple\_full\_binary\_trees  sequence of full trees of the form T=(TT), depths [1..10..1], with/without some brackets at the end, length=49K+  broad\_tree\_with\_deep\_paths  string of the form [TTT...T] of 300 T's, each T being '{{{...}}}' nested 200-fold, length=120K+ |
| 290. 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.  C++1 (3ms)  class Solution {  public:  vector<string> getWords(string str)  {  stringstream ss(str);  vector<string>words;  string token;  while(getline(ss, token, ' '))  words.push\_back(token);  return words;  }  bool wordPattern(string pattern, string str) {  unordered\_map<char, int> Pmap;  unordered\_map<string, int> Smap;  vector<string> tokens = getWords(str);  if(pattern.length()!= tokens.size()) return false;  for(int i=0; i<pattern.length(); ++i)  {  if(Pmap[pattern[i]] != Smap[tokens[i]]) return false;  if(Pmap[pattern[i]]==0 && Smap[tokens[i]]==0)  {  Pmap[pattern[i]] = i+1;  Smap[tokens[i]] = i+1;  }  }  return true;  }  };  C++2 (0ms)  bool wordPattern(string pattern, string str) {  unordered\_map<char, int> PM;  unordered\_map<string, int> SM;  istringstream in(str);  int i=0, n = pattern.size();  for(string word; in>>word; i++)  {  if(i==n || PM[pattern[i]] != SM[word]) return false;  if(!PM[pattern[i]] && !SM[word])  {  PM[pattern[i]] = i+1;  SM[word] = i+1;  }  }  return i==n;  }  Java1 (3ms)  public boolean wordPattern(String pattern, String str) {  String[] words = str.split(" ");  if(pattern.length() != words.length)  return false;  Map addr = new HashMap();  for(Integer i=0; i<words.length; ++i)  {  if(addr.put(pattern.charAt(i), i) != addr.put(words[i], i))  return false;  }  return true;  } |
| 14. Longest Common Prefix  Write a function to find the longest common prefix string amongst an array of strings.  C++1 (9ms) O(mn)  string longestCommonPrefix(vector<string>& strs) {  int wordsLen=strs.size();  int firstWordLen = (wordsLen>0) ? strs[0].size() : 0;  string res;  for(int i=0; i<firstWordLen; ++i)  {  for(int j=1; j<wordsLen; ++j)  {  if(strs[j].size()<i+1 || strs[j][i]!=strs[0][i]) return res;  }  res +=strs[0][i];  }  return res;  }  C++2 (4ms) O(mn)  string longestCommonPrefix(vector<string>& strs) {  string res = "";  bool exist[128] = {0};  for(int j = 0; strs.size()>0; j++)  {  exist[strs[0][j]] = true;  for(int i=0; i<strs.size(); i++)  {  if(j>=strs[i].size() || !exist[strs[i][j]])  return res;  }  res += strs[0][j];  exist[strs[0][j]] = false;  }  return res;  }  C++3 (4ms) O(mn)  string longestCommonPrefix(vector<string>& strs) {  string res = "";  for(int j = 0; strs.size()>0; j++)  {  for(int i=0; i<strs.size(); i++)  {  if(j>=strs[i].size() || (i>0 && strs[i][j] != strs[i-1][j]))  return res;  }  res += strs[0][j];  }  return res;  }  Java1 (1ms) O(mn)  public String longestCommonPrefix(String[] strs) {  if(strs.length == 0) return "";  String res = strs[0];  for(int i=0; i<strs.length; ++i)  {  while(strs[i].indexOf(res)!=0)  res = res.substring(0, res.length()-1);  }  return res;  }  Java2 (5ms) O(mn)  public String longestCommonPrefix(String[] strs) {  StringBuilder res = new StringBuilder();  for(int i=0; strs.length>0; ++i)  {  for(int j=0; j<strs.length; ++j)  {  if(i>=strs[j].length() || (j>0 &&strs[j].charAt(i) != strs[j-1].charAt(i)))  return res.toString();  }  res.append(strs[0].charAt(i));  }  return res.toString();  } |
| 28. Implement strStr()  Implement strStr().  Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.  C++1 (6ms) O(mn)  int strStr(string haystack, string needle) {  int hLen=haystack.length(), nLen=needle.length();  if(nLen>hLen) return -1;  for(int i=0; i<=hLen-nLen; ++i)  if(haystack.substr(i, nLen) == needle)  return i;  return -1;  }  C++2 (4ms) O(mn)  int strStr(string haystack, string needle) {  int last = haystack.size()-needle.size();  for(int i=0; i<=last; i++)  {  bool ok = true;  for(int j=0; j<needle.size(); j++)  {  if(haystack[i+j] != needle[j])  {  ok = false;  break;  }  }  if(ok) return i;  }  return -1;  }  C++3 (8ms) O(mn)  int strStr(string haystack, string needle) {  //if(needle.empty()) return 0;  //if(haystack.empty() || haystack.size()<needle.size()) 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;  }  }  }  Java1 (1ms) O(mn)  public int strStr(String haystack, String needle) {  int hLen=haystack.length(), nLen=needle.length();  if(nLen==0) return 0;  for(int i=0; i<=hLen-nLen; ++i)  if(haystack.substring(i, i+nLen).equals(needle))  return i;  return -1;  }  Java2 (1ms) O(mn)  public int strStr(String haystack, String needle) {  return haystack.indexOf(needle);  } |
| 500. 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.   C++1 (3ms) O(mn)  vector<string> findWords(vector<string>& words) {  char second[] = {'a', 's', 'd', 'f','g','h','j','k', 'l'};  char third[] = {'z','x','c','v','b','n','m'};  int count[128] = {0};  for(char s : second)  count[s]=count[toupper(s)] = 2;  for(char t : third)  count[t]=count[toupper(t)] = 3;  vector<string> res;  for(string word : words)  {  bool isInOneRow = true;  for(int i=1; i<word.length(); ++i)  if(count[word[i]]!= count[word[i-1]])  {  isInOneRow = false;  break;  }  if(isInOneRow)  res.push\_back(word);  }  return res;  }  Python1  def findWords(self, words):  keyboard = ["qwertyuiop", "asdfghjkl", "zxcvbnm"]  keyMap = dict([(K,i) for i in range(3) for K in keyboard[i]])  res =[]  for word in words:  isInOneRow = True  W = word.lower()  for j in range(1,len(W)):  if (keyMap[W[j]] != keyMap[W[j-1]]):  isInOneRow = False  break  if isInOneRow is True:  res.append(word)  return res |
| 520. 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.  C++1 (18ms) O(n)  bool detectCapitalUse(string word) {  char first = word[0];  if(isupper(first))  {  for(int i=2; i<word.size(); ++i)  if(isupper(word[i]) ^ isupper(word[i-1])) return false;  }  else  for(int i=1; i<word.size(); ++i)  if(isupper(word[i])) return false;  return true;  }  Python1  def detectCapitalUse(self, word):  if len(word)==1 :  return True  if word[0].isupper():  return word[1:].isupper() or word[1:].islower()  else:  return word[1:].islower()  Python2  def detectCapitalUse(self, word):  return word.isupper() or word.islower() or word.istitle()  Test cases:   |  |  |  | | --- | --- | --- | | Id | Testcase | Result | | 1 | A | True | | 2 | a | True | | 3 | AB | True | | 4 | ab | True | | 5 | Ab | True | | 6 | aB | False | | 7 | AbA | False | | 8 | aBa | False | | 9 | aBA | False | | 10 | Aba | True | | 11 | abA | False | |  | NULL |  | |  | Other characters than alphbetic |  | |  |  |  | |
| 392. 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 (<=100).  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?  C++1 (69ms) O(m+n)  bool isSubsequence(string s, string t) {  int j=0;  for(int i=0; i<s.length(); ++i,++j)  {  while(j<t.length() && t[j] != s[i]) j++;  if(j>=t.length()) return false;  }  return true;  }  C++2 O(m+n)  bool isSubsequence(string s, string t) {  int sLen = s.length(), i = 0;  if(sLen == 0) return true;  for(char tc : t)  if(tc == s[i] && ++i==sLen)  return true;  return false;  }  C++3 O(m+n)  bool isSubsequence(string s, string t) {  int sLen = s.length(), tLen = t.length();  if(sLen == 0) return true;  int ss=0, se=sLen-1;  for(int ts=0, te=tLen-1; ts<te; ++ts, --te)  if(s[ss] == t[ts] && ++ss>se || s[se]==t[te] && --se<ss)  return true;  return false;  } |
| 1. Integer to Roman   Given an integer, convert it to a roman numeral.  Input is guaranteed to be within the range from 1 to 3999.  C++1  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];  }  Java1  public class Solution {  private String THOUSAND[] = {"", "M", "MM", "MMM"};  private String HUNDURD[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};  private String TEN[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};  private String DIGIT[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};    public String intToRoman(int num) {  return THOUSAND[num/1000] + HUNDURD[(num%1000)/100] + TEN[(num%100)/10] + DIGIT[num%10];  }  } |
| 423. 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"  C++1  string originalDigits(string s) {  int letter[128] = {0};  for(char c : s) letter[c]++;  int digits[10];  digits[0] = letter['z'];  digits[2] = letter['w'];  digits[4] = letter['u'];  digits[6] = letter['x'];  digits[8] = letter['g'];  digits[5] = letter['f'] - digits[4];  digits[7] = letter['v'] - digits[5];  digits[3] = letter['t'] - digits[2] - digits[8];  digits[1] = letter['o'] - digits[0] - digits[2] - digits[4];  digits[9] = letter['i'] - digits[5] - digits[6] - digits[8];  string res;  for(int i=0; i<10; ++i)  if(digits[i] > 0)  res += string(digits[i], '0'+i);  // for(int j=0; j<digit[i]; ++j)  // res += '0'+i;  return res;  } |
| 500. 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.    American keyboard  **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.   C++1  vector<string> findWords(vector<string>& words) {  char second[] = {'a', 's', 'd', 'f','g','h','j','k', 'l'};  char third[] = {'z','x','c','v','b','n','m'};  int count[128] = {0};  for(char s : second)  count[s]=count[toupper(s)] = 2;  for(char t : third)  count[t]=count[toupper(t)] = 3;  vector<string> res;  for(string word : words)  {  bool isInOneRow = true;  for(int i=1; i<word.length(); ++i)  if(count[word[i]]!= count[word[i-1]])  {  isInOneRow = false;  break;  }  if(isInOneRow)  res.push\_back(word);  }  return res;  } |
| 412. Fizz Buzz  Write a program that outputs the string representation of numbers from 1 to *n*.  But for multiples of three it should output “Fizz” instead of the number and for the multiples of five output “Buzz”. For numbers which are multiples of both three and five output “FizzBuzz”.  **Example:**  n = 15,  Return:  [  "1",  "2",  "Fizz",  "4",  "Buzz",  "Fizz",  "7",  "8",  "Fizz",  "Buzz",  "11",  "Fizz",  "13",  "14",  "FizzBuzz"  ]  C++1  vector<string> fizzBuzz(int n) {  vector<string> res(n);  for(int i=1; i<=n; ++i)  {  if(i%15 == 0)  res.push\_back("FizzBuzz");  else if(i%5 == 0)  res.push\_back("Buzz");  else if(i%3 == 0)  res.push\_back("Fizz");  else  res.push\_back(to\_string(i));  } return res;  }  Python1  def fizzBuzz(self, n):  res = []  for i in range(1, n+1):  if i%15==0:  res.append("FizzBuzz")  elif i%5==0:  res.append("Buzz")  elif i%3==0:  res.append("Fizz")  else:  res.append(str(i))  return res |
| 383. 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  C++1 (32ms)  bool canConstruct(string ransomNote, string magazine) {  vector<int> magazLetters(128, 0);  for(char m : magazine)  magazLetters[m]++;  for(char r : ransomNote)  if(--magazLetters[r]<0) return false;  return true;  } |
| 171. Excel Sheet Column Number  Related to question [Excel Sheet Column Title](https://leetcode.com/problems/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  C++1 (6ms)  int titleToNumber(string s) {  int res=0;  for(int i=0; i<s.length(); ++i)  res = res\*26 + toupper(s[i])-'A'+1;  return res;  } |
| 13. Roman to Integer  Given a roman numeral, convert it to an integer.  Input is guaranteed to be within the range from 1 to 3999.  C++1 (44ms)  int romanToInt(string s) {  int sum = 0;  char pre = 0;  int i, len=s.length();  for(i=0; i<len; i++)  {  switch(s[i])  {  case 'I':  sum+=1; pre = 'I'; break;  case 'V':  sum+=5;  if(pre == 'I') sum-=2;  pre = 'V'; break;  case 'X':  sum+=10;  if(pre == 'I') sum-=2;  pre = 'X'; break;  case 'L':  sum+=50;  if(pre == 'X') sum-=20;  pre = 'L'; break;  case 'C':  sum+=100;  if(pre == 'X') sum-=20;  pre = 'C'; break;  case 'D':  sum+=500;  if(pre == 'C') sum-=200;  pre = 'D'; break;  case 'M':  sum+=1000;  if(pre == 'C') sum-=200;  pre = 'M'; break;  }  }  return sum;  } |
| 415. Add Strings  Given two non-negative integers num1 and num2 represented as string, return the sum of num1 and num2.  **Note:**   1. The length of both num1 and num2 is < 5100. 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.   C++1 (3ms)  string addStrings(string num1, string num2) {  int i=num1.length()-1, j=num2.length()-1, carry = 0;  if(i < j) return addStrings(num2, num1);  for(; i>=0 && (j>=0 || carry>0); i--, j--)  {  int a = num1[i]-'0';  int b = (j>=0) ? num2[j]-'0' : 0;  int sum = a+b+carry;  carry = sum/10;  num1[i] = (sum%10) + '0';  }  return (carry) ? "1"+num1 : num1;  }  C++2 (13ms)  string addStrings(string num1, string num2) {  int i=num1.length()-1, j=num2.length()-1, carry = 0;  string res = "";  for(;i>=0 || j>=0 || carry>0; i--, j--)  {  int a = (i>=0) ? num1[i]-'0' : 0;  int b = (j>=0) ? num2[j]-'0' : 0;  int sum = a+b+carry;  carry = sum/10;  res = to\_string(sum%10) + res;  }  return res;  }  C++3 (16ms)  string addStrings(string num1, string num2) {  int i=num1.length()-1, j=num2.length()-1;  int carry = 0;  string res = "";  while(i>=0 || j>=0 || carry>0)  {  carry = carry + ((i>=0) ? num1[i--]-'0' : 0);  carry += (j>=0) ? num2[j--]-'0' : 0;  res = to\_string(carry%10) + res;  carry /= 10;  }  return res;  } |
| 535. Encode and Decode TinyURL  Note: This is a companion problem to the [System Design](https://leetcode.com/problemset/system-design/) problem: [Design TinyURL](https://leetcode.com/problems/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.  C++1 (3ms)  class Solution {  public:  vector<string> res;  // Encodes a URL to a shortened URL.  string encode(string longUrl) {  res.push\_back(longUrl);  return to\_string(res.size()-1);  }  // Decodes a shortened URL to its original URL.  string decode(string shortUrl) {  return res[stoi(shortUrl)];  }  }; |
| 481. 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 "122112" and it contains three 1's, so return 3.  C++1 (9ms)  int magicalString(int n) {  if(n==0) return 0;  queue<int> q;  q.push(2);  int res = 1;  for(int i=2; i<n; ++i)  {  int repeat = q.front();  q.pop();  if(repeat==1) res++;  int input = q.back() ^ 3;  for(int j=0; j<repeat; ++j)  q.push(input);  }  return res;  }  C++2  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');  } |
| 524. 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.   C++1 (103ms)  string findLongestWord(string s, vector<string>& d) {  string res;  int maxL= 0;  sort(d.begin(), d.end());  reverse(d.begin(), d.end());  for(string word : d)  {  int i=0;  for(auto c : s)  {  if(i<word.length() && word[i]==c) i++;  }  if(i== word.length() && i >= maxL)  {  res = word;  maxL = i;  }  }  return res;  } |
| Codility [LongestPassword](https://codility.com/demo/results/trainingNQP69A-DVU/)  You would like to set a password for a bank account. However, there are three restrictions on the format of the password:   * it has to contain only alphanumerical characters (a−z, A−Z, 0−9); * there should be an even number of letters; * there should be an odd number of digits.   You are given a string S consisting of N characters. String S can be divided into *words* by splitting it at, and removing, the spaces. The goal is to choose the longest word that is a valid password. You can assume that if there are K spaces in string S then there are exactly K + 1 words.  For example, given "test 5 a0A pass007 ?xy1", there are five words and three of them are valid passwords: "5", "a0A" and "pass007". Thus the longest password is "pass007" and its length is 7. Note that neither "test" nor "?xy1" is a valid password, because "?" is not an alphanumerical character and "test" contains an even number of digits (zero).  Write a function:  int solution(string &S);  that, given a non-empty string S consisting of N characters, returns the length of the longest word from the string that is a valid password. If there is no such word, your function should return −1.  For example, given S = "test 5 a0A pass007 ?xy1", your function should return 7, as explained above.  Assume that:   * N is an integer within the range [1..200]; * string S consists only of printable ASCII characters and spaces.   In your solution, focus on correctness. The performance of your solution will not be the focus of the assessment.  C++1  #**include** <algorithm>  #**include** <sstream>  #**include** <cstring>  #**include** <vector>  **vector**<**string**> getWords(**string** S)  {  **stringstream** **ss**(S);  **vector**<**string**> res;  **string** token;  **while**(getline(ss, token, ' '))  res.push\_back(token);  **return** res;  }  **int**  (**string** word)  {  **int** alp=0, digit=0;  **for**(**char** w : word)  {  **if**(**isalpha**(w)) alp++;  **else** **if**(**isdigit**(w)) digit++;  **else** **return** 0;  }  **return** (alp%2==0 && digit%2!=0) ? alp+digit : 0;  }  **int** **solution**(**string** &S) {  // write your code in C++14 (g++ 6.2.0)  **vector**<**string**> words = getWords(S);  **int** res = 0;  **for**(**string** word : words)  res = max(res, check(word));  **return** res==0 ? -1 : res;  }  C++2  **int** **check**(**string** word)  {  **int** alp=0, digit=0;  **for**(**char** w : word)  {  **if**(**isalpha**(w)) alp++;  **else** **if**(**isdigit**(w)) digit++;  **else** **return** -1;  }  **return** (alp%2==0 && digit%2!=0) ? alp+digit : -1;  }  **int** **solution**(**string** &S) {  // write your code in C++14 (g++ 6.2.0)  **vector**<**string**> words = getWords(S);  **int** res = -1;  **for**(**string** word : words)  res = max(res, check(word));  **return** res;  }  Testcase  simple  short and simple tests  [▶](https://codility.com/demo/results/trainingJQ54JD-9YM/)  one\_character  one character words  [▶](https://codility.com/demo/results/trainingJQ54JD-9YM/)  one\_word  tests that contains one word only  [▶](https://codility.com/demo/results/trainingJQ54JD-9YM/)  even\_letters  all words have even number of letters  [▶](https://codility.com/demo/results/trainingJQ54JD-9YM/)  odd\_digits  all words have odd number of digits  [▶](https://codility.com/demo/results/trainingJQ54JD-9YM/)  odd\_length  it's sufficient to test validity of characters and if length of word is odd  [▶](https://codility.com/demo/results/trainingJQ54JD-9YM/)  all\_alphanumerical  all words contain only alphanumerical characters  [▶](https://codility.com/demo/results/trainingJQ54JD-9YM/)  extra\_characters  valid passwords joined with some invalid characters  [▶](https://codility.com/demo/results/trainingJQ54JD-9YM/)  large\_random  random tests  [▶](https://codility.com/demo/results/trainingJQ54JD-9YM/)  maximum  biggest possible tests with mixed types of words |
| 616. Add Bold Tag in String  Given a string **s** and a list of strings **dict**, you need to add a closed pair of bold tag <b> and </b> to wrap the substrings in s that exist in dict. If two such substrings overlap, you need to wrap them together by only one pair of closed bold tag. Also, if two substrings wrapped by bold tags are consecutive, you need to combine them.  **Example 1:**  **Input:**  s = "abcxyz123"  dict = ["abc","123"]  **Output:**  "<b>abc</b>xyz<b>123</b>"  **Example 2:**  **Input:**  s = "aaabbcc"  dict = ["aaa","aab","bc"]  **Output:**  "<b>aaabbc</b>c"  **Note:**   1. The given dict won't contain duplicates, and its length won't exceed 100. 2. All the strings in input have length in range [1, 1000].   C++1  int search(string& s, int start, vector<string> words)  {  int len = 0;  for(string w : words)  {  int wl = w.length();  if(start+wl<=s.length() && s.substr(start, wl)==w)  len = max(len, wl);  }  return len;  }    string addBoldTag(string s, vector<string>& dict) {  unordered\_map<char, vector<string>> wdict;  for(string word : dict)  {  wdict[word[0]].push\_back(word);  }    for(int i=0; i<s.length(); ++i)  {  if(wdict.count(s[i]) == 0) continue;  int len = search(s, i, wdict[s[i]]);  if(len == 0) continue;  for(int j=i+1; j<=i+len; ++j)  {  int curLen = search(s, j, wdict[s[j]]);  len = max(len, j-i + curLen);  }  s.insert(i+len, "</b>");  s.insert(i, "<b>");  i += len+7;  }  return s;  } |
|  |

#### Counting Letters

|  |
| --- |
| 58. 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.  C++1 (5ms) O(n)  int lengthOfLastWord(string s) {  int count=1, last=s.find\_last\_not\_of(' ');  if(last<0) return 0;  while(last-->0)  {  if(s[last] != ' ') count++;  else return count;  }  return count;  }  C++2 (4ms) O(n)  int lengthOfLastWord(string s) {  int count = 0;  for(int i= s.size()-1; i>=0; i--)  {  if(isalpha(s[i])) count++;  else if(count == 0) continue;  else return count;  }  return count;  }  Java1 (0ms) O(n)  public int lengthOfLastWord(String s) {  s = s.trim();  int i = s.length()-1;  for(; i>=0; --i)  if(s.charAt(i) == ' ') break; // if(!Character.isLetter(s.charAt(i)))  return s.length()-i-1;  } |
| 424. 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 104.  **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.  C++1 (9ms) O(n)  int characterReplacement(string s, int k) {  int n=s.size(), start=0, max\_same=0;  int letter[128] = {0};  for(int i=0; i<n; ++i)  if((max\_same = max(max\_same, ++letter[s[i]])) < i-start+1-k)  --letter[s[start++]];  return n - start;  } |
| 389. 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.  C++1  char findTheDifference(string s, string t) {  int n = t.length();  for(int i=0; i<n-1; ++i)  t[n-1] ^= s[i] ^ t[i];  return t[n-1];  }  C++2  char findTheDifference(string s, string t) {  int res = 0;  for(int i=0; i<s.length(); ++i)  res ^= s[i] ^ t[i];  return res ^ t[t.length()-1];  } |
| 387. 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.  [Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question.  C++1 (42ms)  int firstUniqChar(string s) {  int count[128] = {0};  for(char c : s)  count[c]++;  for(int i=0; i<s.length(); ++i)  if(count[s[i]] == 1) return i;  return -1;  } |
| 451. Sort Characters By Frequency  Given a string, sort it in decreasing order based on the frequency of characters.  **Example 1:**  **Input:**  "tree"  **Output:**  "eert"  **Explanation:**  'e' appears twice while 'r' and 't' both appear once.  So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.  **Example 2:**  **Input:**  "cccaaa"  **Output:**  "cccaaa"  **Explanation:**  Both 'c' and 'a' appear three times, so "aaaccc" is also a valid answer.  Note that "cacaca" is incorrect, as the same characters must be together.  **Example 3:**  **Input:**  "Aabb"  **Output:**  "bbAa"  **Explanation:**  "bbaA" is also a valid answer, but "Aabb" is incorrect.  Note that 'A' and 'a' are treated as two different characters.  C++1 (12ms)  class Solution {  public:  static bool myComp(pair<char,int> a, pair<char,int> b)  {  return a.second > b.second;  }  string frequencySort(string s) {  vector<pair<char,int>> freq(256, pair<char,int>());  for(char c : s)  {  freq[c].first = c;  freq[c].second++;  }  sort(freq.begin(), freq.end(), myComp);  string res;  for(pair<char, int> pa : freq)  res += string(pa.second, pa.first);  return res;  }  }; |
| 482. 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 characters 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.   C++1 (53ms)  string licenseKeyFormatting(string S, int K) {  int j=1, n=S.length();  string res;  while(j <= n)  {  int i = n-j;  if(S[i] == '-')  {  j++;  continue;  }  if(isalpha(S[i])) S[i] = toupper(S[i]);  res = S[i] + res;  if(++j<=n && res.length()%(K+1)==K) res = '-' + res;  }  return res[0]=='-' ? res.substr(1) : res;  }  C++2 (46ms)  string licenseKeyFormatting(string S, int K) {  string res;  for(int i=S.length()-1; i>=0; --i)  {  if(S[i] != '-')  {  if(isalpha(S[i])) S[i] = toupper(S[i]);  res = S[i] + res;  if(res.length()%(K+1)==K) res = '-' + res;  }  }  return res[0]=='-' ? res.substr(1) : res;  } |
| Codility [SocksLaundering](https://codility.com/demo/results/training2ENN99-XZ8/)  Bob is about to go on a trip. But first he needs to take care of his supply of socks. Each sock has its own color. Bob wants to take as many pairs of clean socks as possible (both socks in the pair should be of the same color).  Socks are divided into two drawers: clean and dirty socks. Bob has time for only one laundry and his washing machine can clean at most K socks. He wants to pick socks for laundering in such a way that after washing he will have a maximal number of clean, same-colored pairs of socks. It is possible that some socks cannot be paired with any other sock, because Bob may have lost some socks over the years.  Bob has exactly N clean and M dirty socks, which are described in arrays C and D, respectively. The colors of the socks are represented as integers (equal numbers representing identical colors).  For example, given four clean socks and five dirty socks:  https://codility-frontend-prod.s3.amazonaws.com/media/task_static/socks_laundering/static/images/auto/bb5550b86d03b06e22142e936bddd02c.png  If Bob's washing machine can clean at most K = 2 socks, then he can take a maximum of three pairs of clean socks. He can wash one red sock and one green sock, numbered 1 and 2 respectively. Then he will have two pairs of red socks and one pair of green socks.  Write a function:  int solution(int K, vector<int> &C, vector<int> &D);  that, given an integer K (the number of socks that the washing machine can clean), two arrays C and D (containing the color representations of N clean and M dirty socks respectively), returns the maximum number of pairs of socks that Bob can take on the trip.  For example, given K = 2, C = [1, 2, 1, 1] and D = [1, 4, 3, 2, 4], the function should return 3, as explained above.  Assume that:   * K is an integer within the range [0..50]; * each element of arrays C, D is an integer within the range [1..50]; * C and D are not empty and each of them contains at most 50 elements.   In your solution, focus on correctness. The performance of your solution will not be the focus of the assessment.  C++1  **int** **solution**(**int** K, **vector**<**int**> &C, **vector**<**int**> &D) {  // write your code in C++14 (g++ 6.2.0)  **int** res = 0;  **vector**<**int**> clean(51);  **vector**<**int**> dirty(51);  **for**(**int** c : C)  clean[c]++;  **for**(**int** d : D)  dirty[d]++;    **for**(**int** i= 1; i<51; ++i)  {  res += clean[i]/2;  **if**(clean[i]%2!=0 && K>0 && dirty[i]>0)  {  res++;  K--;  dirty[i]--;  }  }    **for**(**int** i=1; K>1 && i<51; ++i)  {  **if**(dirty[i] >=2)  {  dirty[i] = min(dirty[i]/2, K/2);  res += dirty[i];  K -= 2\*dirty[i];  }  }  **return** res;  }  Testcase  smallest  smallest possible tests  [▶](https://codility.com/demo/results/training2ENN99-XZ8/)  small\_simple  small simple tests, easy to solve even with heuristic approach  [▶](https://codility.com/demo/results/training2ENN99-XZ8/)  no\_laundry  tests where K = 0  [▶](https://codility.com/demo/results/training2ENN99-XZ8/)  odd\_clean\_with\_odd\_dirty  tests causing solution that's only pairing clean socks of odd count with dirty socks of odd count to fail  [▶](https://codility.com/demo/results/training2ENN99-XZ8/)  saving\_one\_used\_dirty\_check  tests causing solution that's not saving if dirty sock was used to pair with clean sock to fail  [▶](https://codility.com/demo/results/training2ENN99-XZ8/)  all\_dirty  there are no clean socks taken  [▶](https://codility.com/demo/results/training2ENN99-XZ8/)  all\_clean  there are no dirty socks taken after laundry  [▶](https://codility.com/demo/results/training2ENN99-XZ8/)  odd\_even\_dirty\_check  odd and even number of dirty socks which do not match with clean socks  [▶](https://codility.com/demo/results/training2ENN99-XZ8/)  random\_few\_colors  randomly generated tests with only a couple of colors  [▶](https://codility.com/demo/results/training2ENN99-XZ8/)  maximal  maximal possible test cases |
| 551. Student Attendance Record I  You are given a string representing an attendance record for a student. The record only contains the following three characters:   1. **'A'** : Absent. 2. **'L'** : Late. 3. **'P'** : Present.   A student could be rewarded if his attendance record doesn't contain **more than one 'A' (absent)** or **more than two continuous 'L' (late)**.  You need to return whether the student could be rewarded according to his attendance record.  **Example 1:**  **Input:** "PPALLP"  **Output:** True  **Example 2:**  **Input:** "PPALLL"  **Output:** False  C++1  bool checkRecord(string s) {  int a=0, l=0;  for(char c : s)  {  if(c=='L')  {  if(++l>2) return false;  }  else  {  l=0;  if(c=='A'&& ++a>1) return false;  }  }  return true;  } |
| 552. Student Attendance Record II  Given a positive integer **n**, return the number of all possible attendance records with length n, which will be regarded as rewardable. The answer may be very large, return it after mod 109 + 7.  A student attendance record is a string that only contains the following three characters:   1. **'A'** : Absent. 2. **'L'** : Late. 3. **'P'** : Present.   A record is regarded as rewardable if it doesn't contain **more than one 'A' (absent)** or **more than two continuous 'L' (late)**.  **Example 1:**  **Input:** n = 2  **Output:** 8  **Explanation:**  There are 8 records with length 2 will be regarded as rewardable:  "PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"  Only "AA" won't be regarded as rewardable owing to more than one absent times.  **Note:** The value of **n** won't exceed 100,000.  C++1  int checkRecord(int n) {  int mod = 1000000007;  int dp[n+1][2][3];  for(int i=0; i<2; ++i)  for(int j=0; j<3; ++j)  dp[0][i][j] = 1;  for(int p=1; p<=n; ++p)  {  for(int a=0; a<2; ++a)  {  for(int L=0; L<3; ++L)  {  int val = dp[p-1][a][2];  if(a>0) val = (val + dp[p-1][a-1][2]) % mod;  if(L>0) val = (val + dp[p-1][a][L-1]) % mod;  dp[p][a][L] = val;  }  }  }  return dp[n][1][2];  } |
| 657. Judge Route Circle  Initially, there is a Robot at position (0, 0). Given a sequence of its moves, judge if this robot makes a circle, which means it moves back to **the original place**.  The move sequence is represented by a string. And each move is represent by a character. The valid robot moves are R (Right), L(Left), U (Up) and D (down). The output should be true or false representing whether the robot makes a circle.  **Example 1:**  **Input:** "UD"  **Output:** true  **Example 2:**  **Input:** "LL"  **Output:** false  C++1  bool judgeCircle(string moves) {  vector<int> rlud(4);  for(char m : moves){  if(m=='R') rlud[0]++;  else if(m=='L') rlud[1]++;  else if(m=='U') rlud[2]++;  else if(m=='D') rlud[3]++;  }  return rlud[0]==rlud[1] && rlud[2]==rlud[3];  } |
|  |

#### Bit

|  |
| --- |
| 67. Add Binary  Given two binary strings, return their sum (also a binary string).  For example, a = "11" b = "1" Return "100".  C++1 (6ms)  string addBinary(string a, string b) {  string res="";  int carry=0, i=a.length()-1, j=b.length()-1;  while(i>=0 || j>=0 || carry>0)  {  int x = (i>=0) ? (a[i--]-'0') : 0;  int y = (j>=0) ? (b[j--]-'0') : 0;  int sum = x+y+carry;  carry = sum/2;  res = to\_string(sum%2) + res;  }  res.erase(0, res.find\_first\_not\_of('0'));  return (res=="") ? "0" : res;  }  C++2(8ms)  string addBinary(string a, string b) {  string res ="";  int re=0, carry=0, i=a.size()-1, j=b.size()-1;  while(i>=0 && j>=0)  {  re = (int)a[i--] + b[j--] + carry - 96;  carry = re/2;  res = to\_string(re%2) + res;  }  while(i>=0)  {  re = (int)a[i--] + carry - '0';  carry = re/2;  res = to\_string(re%2) + res;  }  while(j>=0)  {  re = (int)b[j--] + carry - '0';  carry = re/2;  res = to\_string(re%2) + res;  }  i=0;  if(carry) res = to\_string(carry) + res;  else  {  while(res[i] == '0') i++;  }  if(res.substr(i, res.size()-i) == "") return "0";  return res.substr(i, res.size()-i);  }  C++3 (4ms)  string addBinary(string a, string b) {  string res ="";  int carry=0, i=a.size()-1, j=b.size()-1;  while(i>=0 || j>=0 || carry==1)  {  carry += i>=0 ? (a[i--] - '0') : 0;  carry += j>=0 ? (b[j--] - '0') : 0;  res = char(carry%2 + '0') + res;  carry /= 2;  }  i = 0;  while(res[i] == '0') i++;  if(res.substr(i, res.size()-i) == "") return "0";  return res.substr(i, res.size()-i);  }  Java1 (15ms)  public String addBinary(String a, String b) {  StringBuilder sb = new StringBuilder();  int i=a.length()-1, j=b.length()-1;  int carry = 0;  while(i>=0 || j>=0 || carry>0)  {  int av = (i>=0) ? a.charAt(i--)-'0' : 0;  int bv = (j>=0) ? b.charAt(j--)-'0' : 0;  int sum = av + bv + carry;  carry = sum/2;  sb.insert(0, sum%2);  }  return (sb.length()==0) ? "0" : sb.toString().replaceFirst("^0+(?!$)","");  }  Java2 (5ms)  public String addBinary(String a, String b) {  StringBuilder sb = new StringBuilder();  int i=a.length()-1, j=b.length()-1;  int carry = 0;  while(i>=0 || j>=0 || carry>0)  {  int av = (i>=0) ? a.charAt(i--)-'0' : 0;  int bv = (j>=0) ? b.charAt(j--)-'0' : 0;  int sum = av + bv + carry;  carry = sum/2;  sb.append(sum%2);  }  return (sb.length()==0) ? "0" : sb.reverse().toString();  } |
| 318. 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.  C++1 (69ms) O(mn)  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)==0)  {  result = max(result, a.second\*b.second);  }  }  }  return result;  }  Java1 (69ms)  public int maxProduct(String[] words) {  Map<Integer, Integer> map = new HashMap<>();  int res = 0;  for(String word : words)  {  int mask = 0, len = word.length();  for(int i=0; i<len; ++i)  {  mask |= 1 << (word.charAt(i)-'a');  }  int longer = (map.containsKey(mask)) ? Math.max(map.get(mask), len) : len;  map.put(mask, longer);  }  for(int key1 : map.keySet())  for(int key2 : map.keySet())  if((key1 & key2) == 0)  res = Math.max(res, map.get(key1)\*map.get(key2));  return res;  } |
|  |

#### Palindrom

|  |
| --- |
| 125. 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.  C++1 (16ms)  bool isPalindrome(string s) {  int i=0, j=s.length()-1;  while(i < j)  {  if(!isalnum(s[i])) i++;  else if(!isalnum(s[j])) j--;  else  {  if(isalpha(s[i])) s[i] = toupper(s[i]);  if(isalpha(s[j])) s[j] = toupper(s[j]);  if(s[i]!=s[j]) return false;  i++;  j--;  }  }  return true;  }  C++2 (14ms)  bool isPalindrome(string s) {  int i=0, j=s.size()-1;  while(i<j)  {  while(i<j && !isalnum(s[i])) ++i;  while(i<j && !isalnum(s[j])) --j;  if(isalpha(s[i]) ^ isalpha(s[j])) return false;  int d = abs(s[i] - s[j]);  if(d!=0 && d!=32) return false;  ++i;  --j;  }  return true;  }  Java1 (9ms)  public boolean isPalindrome(String s) {  for(int i=0, j=s.length()-1; i<j; i++, j--)  {  while(i<j && !Character.isLetterOrDigit(s.charAt(i)))  i++;  while(i<j && !Character.isLetterOrDigit(s.charAt(j)))  j--;  if(i==j)  return true;  char left = Character.toUpperCase(s.charAt(i));  char right = Character.toUpperCase(s.charAt(j));  if(left != right)  return false;  }  return true;  } |
| 409. 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:  "abccccdd"  Output:  7  Explanation:  One longest palindrome that can be built is "dccaccd", whose length is 7.  C++1 (3ms)  int longestPalindrome(string s) {  int letter[128] = {0}, n = s.length();  for(int i=0; i<n; ++i)  letter[s[i]]++;  int res = 0, odd = 0;  for(int i=65; i<=122; ++i)  {  res += letter[i];  if(letter[i]%2 != 0)  odd++;  }  return (odd>1) ? res-odd+1 : res;  }  Java1 (6ms)  int longestPalindrome(string s) {  int res = 0;  int count[128] = {0};  for(char c : s)  count[c]++;  int hasOdd = 0;  for(int i='A'; i<='z'; ++i)  {  if(count[i]%2==0) res += count[i];  else  {  res += count[i]-1;  hasOdd = 1;  }  }  return res + hasOdd;  } |
| 516. 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".  C++1 (49ms)  int longestPalindromeSubseq(string s) {  int n = s.length();  if(n==0) return 0;  vector<vector<int>> dp(n, vector<int>(n));  for(int i=n-1; i>=0; --i)  {  dp[i][i] = 1;  for(int j=i+1; j<n; ++j)  dp[i][j] = (s[i]==s[j]) ? dp[i+1][j-1] + 2  : max(dp[i+1][j], dp[i][j-1]);  }  return dp[0][n-1];  }  C++2 (TLE)  class Solution {  public:  int RescursivePalindrome(string s, vector<vector<int>>& dp, int start, int end)  {  if(dp[start][end] != 0) return dp[start][end];  if(start > end) return 0;  if(start == end) return 1;  dp[start][end] = (s[start] == s[end]) ? RescursivePalindrome(s, dp, start+1, end-1) + 2  : max(RescursivePalindrome(s, dp, start+1, end), RescursivePalindrome(s, dp, start, end-1));  return dp[start][end];  }  int longestPalindromeSubseq(string s) {  int n = s.length();  if(n==0) return 0;  vector<vector<int>> dp(n, vector<int>(n));  return RescursivePalindrome(s, dp, 0, n-1);  }  };  Testcases:   |  |  |  | | --- | --- | --- | | Id | Testcase | Result | | 1 | [] | 0 | | 2 | [a] | 1 | | 3 | "fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffgggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg" | 494 | |
| Codility [StrSymmetryPoint](https://codility.com/demo/results/trainingE7GKSA-Y65/)  Task description  Write a function:  int solution(string &S);  that, given a string S, returns the index (counting from 0) of a character such that the part of the string to the left of that character is a reversal of the part of the string to its right. The function should return −1 if no such index exists.  *Note:* reversing an empty string (i.e. a string whose length is zero) gives an empty string.  For example, given a string:  "racecar"  the function should return 3, because the substring to the left of the character "e" at index 3 is "rac", and the one to the right is "car".  Given a string:  "x"  the function should return 0, because both substrings are empty.  Assume that:   * the length of S is within the range [0..2,000,000].   Complexity:   * expected worst-case time complexity is O(length(S)); * expected worst-case space complexity is O(1) (not counting the storage required for input arguments).   C++1  **int** **solution**(**string** &S) {  // write your code in C++14 (g++ 6.2.0)  **int** n = S.length();  **if**(n%2 ==0) **return** -1;  **int** i=0;  **for**(**int** j=n-1; i<j; ++i, --j)  **if**(S[i] != S[j]) **return** -1;  **return** i;  }  Testcase  extreme\_empty\_or\_one  empty or one character strings  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  symmetric  short symmetric strings  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  even  even length or symmetric strings  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  three\_chars  3 characters (multiple runs)  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  letters\_a  letters 'a' only  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  alphabet\_symmetric  nontrivial symmetry, N = 51  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  nonsymmetric\_inside  mismatch close to the middle, N = 43  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  nonsymmetric\_outside  mismatch close to the ends, N = 43  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  large\_nonsymmetric  nonsymmetric string, N = 100k+ + [aba]  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  large\_symmetric1  symmetric string, N=100k  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  large\_symmetric2  symmetric string, N=200k  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  big\_symmetric3  symmetric string, N=1M+  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  big\_nonsymmetric  nonsymmetric string, N = ~1M  [▶](https://codility.com/demo/results/trainingE7GKSA-Y65/)  extreme\_size  N = ~2M |
| 131. Palindrome Partitioning  Given a string *s*, partition *s* such that every substring of the partition is a palindrome.  Return all possible palindrome partitioning of *s*.  For example, given *s* = "aab", Return  [  ["aa","b"],  ["a","a","b"]  ]  C++1  class Solution {  public:  bool isPalindrome(string& s, int start, int end)  {  while(start<end)  {  if(s[start++]!= s[end--])  return false;  }  return true;  }  void PPHelper(vector<vector<string>>& res, vector<string>& arow, string& s, int start, int len)  {  if(start==len)  {  res.push\_back(arow);  }  for(int i=start; i<len; ++i)  {  if(isPalindrome(s, start, i))  {  arow.push\_back(s.substr(start, i-start+1));  PPHelper(res, arow, s, i+1, len);  arow.pop\_back();  }  }  }  vector<vector<string>> partition(string s) {  vector<vector<string>> res;  int len = s.length();  if(len<=0) return res;  vector<string> arow;  PPHelper(res, arow, s, 0, len);  return res;  }  };  C++2 (the speed is not imporved! No need memory?)  class Solution {  public:  bool isPalindrome(string& s)  {  int i=0, j=s.length()-1;  while(i<j)  if(s[i++] != s[j--]) return false;  return true;  }  void dfs(vector<vector<string>>& res, vector<string>& aRow, string & s, vector<vector<int>>& visitPali, int start)  {  if(start >= s.length())  {  res.push\_back(aRow);  }  for(int len=1; len<=s.length()-start; ++len)  {  string sub = s.substr(start, len);  if(visitPali[start][start+len]==2)  {  aRow.push\_back(sub);  dfs(res, aRow, s, visitPali, start+len);  aRow.pop\_back();  }  else if(visitPali[start][start+len]==0)  {  if(isPalindrome(sub))  {  visitPali[start][start+len] = 2;  aRow.push\_back(sub);  dfs(res, aRow, s, visitPali, start+len);  aRow.pop\_back();  }  else  visitPali[start][start+len] = 1;  }  }  }  vector<vector<string>> partition(string s) {  vector<vector<string>> res;  vector<string> aRow;  vector<vector<int>> visitPali(s.length(), vector<int>(s.length()+1));  for(int i=0;i<s.length(); ++i)  visitPali[i][1] = 2;  dfs(res, aRow, s, visitPali, 0);  return res;  }  };  C++3  class Solution {  public:  void PPHelper(vector<vector<string>>& res, vector<string>& arow, string& s, int start, int len)  {  if(start==len)  {  res.push\_back(arow);  }  string str = "";  for(int i=start; i<len; ++i)  {  str+=s[i];  if(str == string(str.rbegin(), str.rend()))  {  arow.push\_back(str);  PPHelper(res, arow, s, i+1, len);  arow.pop\_back();  }  }  }  vector<vector<string>> partition(string s) {  vector<vector<string>> res;  int len = s.length();  vector<string> arow;  PPHelper(res, arow, s, 0, len);  return res;  }  }; |
| 9. Palindrome Number  Determine whether an integer is a palindrome. Do this without extra space.  C++ (222ms)  bool isPalindrome(int x) {  int n = x;  long res = 0;  while(n>0)  {  res = res\*10 + n%10;  n/=10;  }  return res==x;  }  C++2 |
| 479. Largest Palindrome Product  Find the largest palindrome made from the product of two n-digit numbers.  Since the result could be very large, you should return the largest palindrome mod 1337.  **Example:**  Input: 2  Output: 987  Explanation: 99 x 91 = 9009, 9009 % 1337 = 987  **Note:**  The range of n is [1,8].  C++1  int largestPalindrome(int n) {  int res[] = {9, 987, 123, 597, 677, 1218, 877, 475};  return res[n-1];  } |
| 647. Palindromic Substrings  Given a string, your task is to count how many palindromic substrings in this string.  The substrings with different start indexes or end indexes are counted as different substrings even they consist of same characters.  **Example 1:**  **Input:** "abc"  **Output:** 3  **Explanation:** Three palindromic strings: "a", "b", "c".  **Example 2:**  **Input:** "aaa"  **Output:** 6  **Explanation:** Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".  **Note:**   1. The input string length won't exceed 1000.   C++1  class Solution {  public:  int countFrom(int i, int j, string & s)  {  int c = 0;  while(i>=0 && j<s.length() && s[i--]==s[j++])  c++;  return c;  }    int countPalindrom(int i, string& s)  {  int c = 0;  c += countFrom(i-1, i+1, s);  c += countFrom(i, i+1, s);  return c;  }    int countSubstrings(string s) {  int res = 0, n = s.length();  for(int i=1; i<n-1; ++i)  res += countPalindrom(i, s);  return n+res + ((n>0 && s[0]==s[1]) ? 1 : 0);  }  };  Java1  public int countSubstrings(String s) {  int res=0, n=s.length();  for(int i=1; i<n-1; ++i){  int temp = 0;  int left=i-1, right=i+1;  while(left>=0 && right<n && s.charAt(left--)==s.charAt(right++))  temp++;  left=i;  right=i+1;  while(left>=0 && right<n && s.charAt(left--)==s.charAt(right++))  temp++;  res += temp;  }  return n+res + ((n>1 && s.charAt(0)==s.charAt(1)) ? 1 : 0);  }  Java2  public int countSubstrings(String s) {  int res=0, n=s.length();  char[] cs = s.toCharArray();  for(int i=1; i<n-1; ++i){  int temp = 0;  int left=i-1, right=i+1;  while(left>=0 && right<n && cs[left--]==cs[right++])  temp++;  left=i;  right=i+1;  while(left>=0 && right<n && cs[left--]==cs[right++])  temp++;  res += temp;  }  return n+res + ((n>1 && cs[0]==cs[1]) ? 1 : 0);  } |
|  |

#### Parentheses

|  |
| --- |
| 22. 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:  [  "((()))",  "(()())",  "(())()",  "()(())",  "()()()"  ]  C++1 (3ms)  public:  void GetParenthesis(vector<string>& res, string aForm, int open, int close)  {  if(open==0 && close==0) res.push\_back(aForm);  else if(close>0)  {  if(open>0)  {  aForm.push\_back('(');  GetParenthesis(res, aForm, open-1, close);  aForm.pop\_back();  }  if(close > open)  {  aForm.push\_back(')');  GetParenthesis(res, aForm, open, close-1);  aForm.pop\_back();  }  }  }  vector<string> generateParenthesis(int n) {  vector<string> res;  string aForm;  GetParenthesis(res, aForm, n, n);  return res;  }  };  C++2 (0ms)  class Solution {  public:  vector<string> res;  void Helper(string str, int left, int right)  {  if(left>0) Helper(str+"(", left-1, right);  if(right>left) Helper(str+")", left, right-1);  if(right==0) res.push\_back(str);  }  vector<string> generateParenthesis(int n) {  Helper("", n, n);  return res;  }  };  Java1 (2ms)  public class Solution {    private List<String> res;    public List<String> generateParenthesis(int n) {  res = new ArrayList<String>();  GPhelper("", n, n);  return res;  }    private void GPhelper(String arow, int left, int right)  {  if(left > 0) GPhelper(arow+"(", left-1, right);  if(right > left) GPhelper(arow+")", left, right-1);  if(right == 0) res.add(arow);  }  } |
| 241. 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]  C++1 (9ms)  vector<int> diffWaysToCompute(string input) {  vector<int> res;  for(int i=0; i<input.length(); ++i)  {  char c = input[i];  if(ispunct(c))  for(auto a : diffWaysToCompute(input.substr(0, i)))  for(auto b : diffWaysToCompute(input.substr(i+1)))  res.push\_back(c=='+' ? a+b : (c=='-') ? a-b : a\*b);  }  return res.size()==0 ? vector<int>{stoi(input)} : res;  }  Java1  public List<Integer> diffWaysToCompute(String input) {  List<Integer> res = new LinkedList<Integer>();  for(int i=0; i<input.length(); ++i)  {  char ch = input.charAt(i);  if(!Character.isDigit(ch))  {  String left = input.substring(0, i);  String right = input.substring(i+1, input.length());  for(int a : diffWaysToCompute(left))  for(int b : diffWaysToCompute(right))  res.add((ch == '+') ? a+b :  (ch == '-') ? a-b :  a\*b);  }  }  if(res.size() == 0)  res.add(Integer.parseInt(input));  return res;  } |
| 394. 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".  C++1 (0ms) O(n)  string decodeString(string s) {  int n = 0;  string word="";  stack<int> repeat;  stack<string> words;  for(auto c : s)  {  if(isdigit(c))  n = 10\*n + (c-'0');  else if(c=='[')  {  repeat.push(n);  n=0;  words.push(word);  word.clear();  }  else if(c==']')  {  int rep = repeat.top();  repeat.pop();  while(rep-- > 0)  words.top() += word;  word = words.top();  words.pop();  }  else  word += c;  }  return words.empty() ? word : words.top();  } |
| 553. Optimal Division  Given a list of **positive integers**, the adjacent integers will perform the float division. For example, [2,3,4] -> 2 / 3 / 4.  However, you can add any number of parenthesis at any position to change the priority of operations. You should find out how to add parenthesis to get the **maximum** result, and return the corresponding expression in string format. **Your expression should NOT contain redundant parenthesis.**  **Example:**  **Input:** [1000,100,10,2]  **Output:** "1000/(100/10/2)"  **Explanation:**  1000/(100/10/2) = 1000/((100/10)/2) = 200  However, the bold parenthesis in "1000/(**(**100/10**)**/2)" are redundant,  since they don't influence the operation priority. So you should return "1000/(100/10/2)".  Other cases:  1000/(100/10)/2 = 50  1000/(100/(10/2)) = 50  1000/100/10/2 = 0.5  1000/100/(10/2) = 2  **Note:**   1. The length of the input array is [1, 10]. 2. Elements in the given array will be in range [2, 1000]. 3. There is only one optimal division for each test case.   C++1  string optimalDivision(vector<int>& nums) {  int n = nums.size();  string res = to\_string(nums[0]);  if(n==1) return res;  if(n==2) return res + "/" + to\_string(nums[1]);  res += "/(" + to\_string(nums[1]);  for(int i=2; i<n; ++i)  res += "/" + to\_string(nums[i]);  return res + ")";  } |
|  |
| 227. 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.  C++1  int calculate(string s) {  stack<int> nums;  char operators = '+';  int res=0, sign = 1;  for(char c : s)  {  if(isdigit(c))  res = res\*10 + (c-'0');  else if(c != ' ')  {  nums.push(sign\*res);  if(operators=='\*' || operators=='/')  {  int b = nums.top();  nums.pop();  int a = nums.top();  nums.pop();  nums.push((operators=='\*') ? a\*b : a/b);  }  operators = c;  if(c=='-')  sign=-1;  else  sign = 1;  res=0;  }  }  res = sign\*res;  if(operators=='\*' || operators=='/')  {  res = (operators=='\*') ? nums.top()\*res : nums.top()/res;  nums.pop();  }  while(!nums.empty())  {  res += nums.top();  nums.pop();  }  return res;  }  C++2  int calculate(string s) {  stack<int>pos;  s = s+"+";  int num=0;  char sign = '+';  for(char c : s)  {  if(isdigit(c))  {  num = num\*10 + (c-'0');  }  else if(!isspace(c))  {  switch(sign)  {  case '+':  pos.push(num);  break;  case '-':  pos.push(-num);  break;  case '\*':  num \*=pos.top();  pos.pop();  pos.push(num);  break;  case '/':  num =pos.top()/num;  pos.pop();  pos.push(num);  break;  }  num = 0;  sign = c;  }  }  while(!pos.empty())  {  num +=pos.top();  pos.pop();  }  return num;  } |
|  |

#### Reverse

|  |
| --- |
| 344. Reverse String  Write a function that takes a string as input and returns the string reversed.  **Example:** Given s = "hello", return "olleh".  C++1  string reverseString(string s) {  int i = 0, j = s.size() - 1;  while(i < j){  swap(s[i++], s[j--]);  }  return s;  }  C++2  string reverseString(string s) {  int len = s.size();  for(int i=0; i<len/2; i++)  swap(s[i], s[len-i-1]);  return s;  }  Java1  public String reverseString(String s) {  int n = s.length();  StringBuilder sb= new StringBuilder();  char[] c = s.toCharArray();  for(int i=n-1; i>=0; --i)  sb.append(c[i]);  return sb.toString();  }  Python1  def reverseString(self, s):  return s[::-1] |
| 541. 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]   C++1 O(kn)  class Solution {  public:  void reverseStrFromTo(string& s, int i, int j)  {  while(i < j)  swap(s[i++], s[j--]);  }  string reverseStr(string s, int k) {  int n = s.length();  for(int i=0; i<n; i+=2\*k)  {  int j = i+k-1;  while(j>=n) j--;  reverseStrFromTo(s, i, j);  }  return s;  }  }; |
| 557. Reverse Words in a String III  Given a string, you need to reverse the order of characters in each word within a sentence while still preserving whitespace and initial word order.  **Example 1:**  **Input:** "Let's take LeetCode contest"  **Output:** "s'teL ekat edoCteeL tsetnoc"  **Note:** In the string, each word is separated by single space and there will not be any extra space in the string.  C++1  class Solution {  public:  void swapStr(string &s, int i, int j)  {  while(i<j) swap(s[i++], s[j--]);  }  string reverseWords(string s) {  int i=0, j=0, n = s.length();  while(i<n-1)  {  while(j+1<n && s[j+1]!=' ') j++;  swapStr(s, i, j);  i=j=j+2;  }  return s;  }  };  Python1  def reverseWords(self, s):  listS = s.split();  for i in range(len(listS)):  listS[i] = listS[i][::-1]  return " ".join(listS) |
| 151. 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.](https://leetcode.com/problems/reverse-words-in-a-string/)  **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.   C++1  void reverseWords(string &s) {  reverse(s.begin(), s.end());  int cur=0;  for(int i=0; i<s.length(); ++i)  {  if(s[i] != ' ')  {  if(cur!=0) s[cur++] = ' ';  int j=i;  while(j<s.length() && s[j]!=' ') s[cur++] = s[j++];  int L=cur-(j-i), R=cur-1;  while(L < R) swap(s[L++], s[R--]);  i = j;  }  }  s.resize(cur);  } |
| 7. Reverse Integer  Reverse digits of an integer.  **Example1:** x = 123, return 321 **Example2:** x = -123, return -321  [click to show spoilers.](https://leetcode.com/problems/reverse-integer/?tab=Description)  **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.  **Note:** The input is assumed to be a 32-bit signed integer. Your function should **return 0 when the reversed integer overflows**.  C++1 (15ms)  int reverse(int x) {  int res = 0;  while(x!=0)  {  int pre = res;  res = 10\*res + x%10; //negative number mod a number equals negative  if((res-x%10)/10 != pre) return 0;  x/=10;  }  return res;  } |
|  |

#### SubString

|  |
| --- |
| 459. 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.)  C++1 (36ms)  bool repeatedSubstringPattern(string str) {  int len = str.length();  for(int i=len/2; i>=1; --i)  {  if(len%i == 0)  {  int m = len/i;  string sub = str.substr(0,i);  string subSum = "";  for(int j=0; j<m; ++j)  subSum += sub;  if(subSum == str) return true;  }  }  return false;  }  C++2 (1682ms)  class Solution {  public:  bool isNRepeatedPattern(string str, int n)  {  for(int i=n; i<str.length(); ++i)  if((str[i] != str[i%n])) return false;  return true;  }  bool repeatedSubstringPattern(string str) {  int len = str.length();  for(int i=2; i<=len; ++i)  if(isNRepeatedPattern(str, len/i)) return true;  return false;  }  }; |
| 467. 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 "zab" in the string s.  C++1 (13ms)  int findSubstringInWraproundString(string p) {  if(p.empty()) return 0;  vector<int> dp(128);  dp[p[0]] = 1;  for(int i=1, maxL = 1; i<p.length(); ++i)  {  maxL = (p[i]-p[i-1]==1 || p[i]-p[i-1]==-25) ? maxL+1 : 1;  dp[p[i]] = max(dp[p[i]], maxL);  }  return accumulate(dp.begin()+'a', dp.begin()+'z'+1, 0);  } |
| 521. Longest Uncommon Subsequence I  Given a group of two strings, you need to find the longest uncommon subsequence of this group of two strings. The longest uncommon subsequence is defined as the longest subsequence of one of these strings and this subsequence should not be **any** subsequence of the other strings.  A **subsequence** is a sequence that can be derived from one sequence by deleting some characters without changing the order of the remaining elements. Trivially, any string is a subsequence of itself and an empty string is a subsequence of any string.  The input will be two strings, and the output needs to be the length of the longest uncommon subsequence. If the longest uncommon subsequence doesn't exist, return -1.  **Example 1:**  **Input:** "aba", "cdc"  **Output:** 3  **Explanation:** The longest uncommon subsequence is "aba" (or "cdc"),  because "aba" is a subsequence of "aba",  but not a subsequence of any other strings in the group of two strings.  **Note:**   1. Both strings' lengths will not exceed 100. 2. Only letters from a ~ z will appear in input strings.   C++1  int findLUSlength(string a, string b) {  return a==b ? -1 : max(a.length(), b.length());  } |
| 522. Longest Uncommon Subsequence II  Given a list of strings, you need to find the longest uncommon subsequence among them. The longest uncommon subsequence is defined as the longest subsequence of one of these strings and this subsequence should not be **any** subsequence of the other strings.  A **subsequence** is a sequence that can be derived from one sequence by deleting some characters without changing the order of the remaining elements. Trivially, any string is a subsequence of itself and an empty string is a subsequence of any string.  The input will be a list of strings, and the output needs to be the length of the longest uncommon subsequence. If the longest uncommon subsequence doesn't exist, return -1.  **Example 1:**  **Input:** "aba", "cdc", "eae"  **Output:** 3  **Note:**   1. All the given strings' lengths will not exceed 10. 2. The length of the given list will be in the range of [2, 50].   C++1  class Solution {  public:  bool isSubsquence(string x, string y)  {  int i=0;  for(int j=0; i<x.length() && j<y.length(); ++j)  if(x[i] == y[j]) i++;  return i==x.length();  }  int findLUSlength(vector<string>& strs) {  int res = -1, n = strs.size();  for(int i=0, j; i<n; ++i)  {  for(j=0; j<n; ++j)  {  if(i==j) continue;  if(isSubsquence(strs[i], strs[j])) break;  }  if(j==n)  res = max(res, (int)strs[i].length());  }  return res;  }  };  C++2  class Solution {  public:  bool isSubsquence(string x, string y)  {  int i=0;  for(int j=0; i<x.length() && j<y.length(); ++j)  if(x[i] == y[j]) i++;  return i==x.length();  }  int findLUSlength(vector<string>& strs) {  int n = strs.size();  sort(strs.begin(), strs.end(), [](string x, string y){  return x.length() > y.length();  });  for(int i=0, j; i<n; ++i)  {  for(j=0; j<n; ++j)  {  if(i==j) continue;  if(isSubsquence(strs[i], strs[j])) break;  }  if(j==n)  return strs[i].length();  }  return -1;  }  }; |
| 395. 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.  C++1  int longestSubstring(string s, int k) {  if(s.length() < k) return 0;  int letters[26] = {0};  for(char c : s)  letters[c-'a']++;  int mid = 0;  while(mid<s.length() && letters[s[mid]-'a']>=k) mid++;  if(mid==s.length()) return s.length();  int left = longestSubstring(s.substr(0,mid), k);  int right = longestSubstring(s.substr(mid+1), k);  return max(left, right);  } |
| 567. Permutation in String  Given two strings **s1** and **s2**, write a function to return true if **s2** contains the permutation of **s1**. In other words, one of the first string's permutations is the **substring** of the second string.  **Example 1:**  **Input:**s1 = "ab" s2 = "eidbaooo"  **Output:**True  **Explanation:** s2 contains one permutation of s1 ("ba").  **Example 2:**  **Input:**s1= "ab" s2 = "eidboaoo"  **Output:** False  **Note:**   1. The input strings only contain lower case letters. 2. The length of both given strings is in range [1, 10,000].   C++1  class Solution {  public:  bool hasSameLetters(string& sub, vector<int>& ls)  {  vector<int> ls2(26,0);  for(char c : sub)  ls2[c-'a']++;  for(int i=0; i<26; ++i)  if(ls[i] != ls2[i]) return false;  return true;  }  bool checkInclusion(string s1, string s2) {  vector<int> ls(26,0);  for(char c : s1)  ls[c-'a']++;  int subS= s1.length();  for(int i=0; i+subS<=s2.size(); ++i)  {  string sub = s2.substr(i, subS);  if(hasSameLetters(sub, ls)) return true;  }  return false;  }  }; |
| 306. 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?  C++1  class Solution {  public:  string add(string& n1, string& n2)  {  string res;  int carry=0, i=n1.length()-1, j=n2.length()-1;  while(i>=0 || j>=0 || carry>0)  {  int a = (i>=0) ? n1[i--]-'0' : 0;  int b = (j>=0) ? n2[j--]-'0' : 0;  int sum = a+b+carry;  carry = sum/10;  res = to\_string(sum%10) + res;  }  return res;  }  bool isAdditive(string n1, string n2, string rest)  {  if(n1.length()>1 && n1[0]=='0' || n2.length()>1 && n2[0]=='0') return false;  string sum = add(n1, n2);  if(sum == rest) return true;  //cause sum!=rest, so if sum.length()==res.length(), should return false  if(sum.length()>=rest.length() || sum != rest.substr(0,sum.length())) return false;  return isAdditive(n2, sum, rest.substr(sum.length()));  }  bool isAdditiveNumber(string num) {  int n = num.length();  for(int i=1; i<=n/2; ++i)  for(int j=i+1; max(i, j-i)<=n-j; ++j)  if(isAdditive(num.substr(0, i), num.substr(i, j-i), num.substr(j)))  return true;  return false;  }  };  C++2  class Solution {  public:  string stringAdd(string num1, string num2)  {  int n1=num1.size(), n2=num2.size();  int sum=0, carry=0, i=n1-1, j=n2-1;  string res;  while(i>=0 || j>=0 || carry>0)  {  int a = (i>=0) ? num1[i--]-'0' : 0;  int b = (j>=0) ? num2[j--]-'0' : 0;  sum = a+b+carry;  carry = sum/10;  res += to\_string(sum%10);  }  reverse(res.begin(), res.end());  return res;  }  bool isAdditive(string num1, string num2, string rest)  {  if(num1.size()>1 && num1[0]=='0' || num2.size()>1 && num2[0]=='0') return false;  string sum = stringAdd(num1, num2);  if(sum==rest) return true;  if(sum.size()>=rest.size() || sum != rest.substr(0, sum.size())) return false;  return isAdditive(num2, sum, rest.substr(sum.size()));  }  bool isAdditiveNumber(string num) {  int n = num.size();  for(int i=1; i<=n/2; ++i)  {  for(int j=1; max(i,j)<=n-i-j; ++j)  {  if(isAdditive(num.substr(0,i), num.substr(i,j), num.substr(i+j))) return true;  }  }  return false;  }  }; |
|  |

#### IP

|  |
| --- |
| 93. 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)  C++1  class Solution {  public:  bool isValid(string& sub)  {  if(sub.size()>3 || sub.size()>1 && sub[0]=='0' || stoi(sub)>255) return false;  return true;  }  vector<string> restoreIpAddresses(string s) {  vector<string> res;  int n = s.length();  for(int i=1; i<4 && i<=n-3;++i)  {  for(int j=i+1; j<i+4 && j<=n-2; ++j)  {  for(int k=j+1; k<j+4 && k<=n-1; ++k)  {  string s1=s.substr(0,i), s2=s.substr(i,j-i), s3=s.substr(j,k-j), s4=s.substr(k);  if(isValid(s1) && isValid(s2) && isValid(s3) && isValid(s4))  res.push\_back(s1+"."+s2+"."+s3+"."+s4);  }  }  }  return res;  }  }; |
| 468. Validate IP Address  Write a function to check whether an 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, leading zeros in the IPv4 is invalid. For example, the address 172.16.254.01 is invalid.  **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 valid 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, extra leading zeros in the IPv6 is also invalid. For example, the address 02001:0db8:85a3:0000:0000:8a2e:0370:7334 is invalid.  **Note:** You may assume there is no extra space or 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.  C++1  class Solution {  public:  bool isIPv4(string IP)  {  stringstream ss(IP);  string block;  for(int i=0; i<4; ++i)  if(!getline(ss, block, '.') || !isValidIPv4Block(block))  return false;  return ss.eof() ? true : false;  }  bool isValidIPv4Block(string block)  {  if(!block.empty() && block.size()<=3)  {  int num = 0;  for(int i=0; i<block.size(); ++i)  {  if(!isdigit(block[i]) || i==0 && block[i]=='0' && block.size()>1)  return false;  num = num\*10 + (block[i] - '0');  }  return num <= 255;  }  return false;  }  bool isIPv6(string IP)  {  stringstream ss(IP);  string block;  for(int i=0; i<8; ++i)  if(!getline(ss, block, ':') || !isValidIPv6Block(block))  return false;  return ss.eof() ? true : false;  }    const string IPv6Str = "0123456789abcdefABCDEF";  bool isValidIPv6Block(string block)  {  if(block.size()>0 && block.size()<=4)  {  for(int i=0; i<block.size(); ++i)  if(IPv6Str.find(block[i])==string::npos)  return false;  return true;  }  return false;  }  string validIPAddress(string IP) {  if(IP.substr(0,4).find('.') != string::npos && isIPv4(IP))  return "IPv4";  else if(IP.substr(0,5).find(':') != string::npos && isIPv6(IP))  return "IPv6";  return "Neither";  }  }; |
|  |

#### Operation

|  |
| --- |
| 43. Multiply Strings  Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2.  **Note:**   1. The length of both num1 and num2 is < 110. 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.   C++1  class Solution {  public:  string multiplyOneDigit(string num2, int n)  {  string res = "";  int carry=0;  for(int i=num2.length()-1; i>=0; --i)  {  int a =num2[i]-'0';  int mult = a\*n +carry;  carry = mult/10;  res += to\_string(mult%10);  }  return carry>0 ? res + to\_string(carry) : res;  }  string add(string pre, string cur, int len)  {  string res = pre.substr(0, len);  int i=len, j=0, carry=0;  int pLen= pre.length(), cLen = cur.length();  while(i<pLen || j<cLen || carry>0)  {  int a = i<pLen ? pre[i++]-'0' : 0;  int b = j<cLen ? cur[j++]-'0' : 0;  int sum = a+b+carry;  carry = sum/10;  res += to\_string(sum%10);  }  return res;  }  string multiply(string num1, string num2) {  string res="";  if(num1.empty()) return num2;  if(num2.empty()) return num1;  if(num1[0]=='0' || num2[0]=='0') return "0";  int n = num1.length();  for(int i=n-1; i>=0; --i)  {  string str = multiplyOneDigit(num2, num1[i]-'0');  res = add(res, str, n-i-1);  }  reverse(res.begin(), res.end());  return res;  }  };  C++2  string multiply(string num1, string num2) {  long ns1=num1.size(), ns2=num2.size();  if(ns1==0 || ns2==0) return "0";  int n1, n2, sum, carry;  vector<int> dp(ns1+ns2, 0);  for(long i=0; i<ns1; ++i)  {  n1 = num1[ns1-i-1] - '0';  carry=0;  for(long j=0; j<ns2; ++j)  {  n2 = num2[ns2-j-1] - '0';  sum = n1\*n2 + dp[i+j] + carry;  carry = sum/10;  dp[i+j] = sum%10;  }  if(carry>0) dp[i+ns2] += carry;  }  long start=ns1+ns2-1;  while(start>=0 && dp[start]==0) start--;  if(start==-1) return "0";  string res = "";  for(long i=start; i>=0; --i) res += '0' + dp[i];  return res;  } |
|  |
|  |
|  |

#### Conversion

|  |
| --- |
| 6. 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".  C++1  string convert(string s, int numRows) {  if(numRows<=1 || s.length()<=numRows) return s;  vector<string> buffer(numRows);  int sign=1, row=0;  for(char c : s)  {  buffer[row]+=c;  row +=sign;  if(row==0 || row==numRows-1) sign = -sign;  }  string res = "";  for(string s : buffer)  res += s;  return res;  }  C++2  string convert(string s, int numRows) {  if (numRows <= 1 || s.size()<=numRows)  return s;  vector<string> v(numRows, "");  int row=0, step;  for(int i=0; i<s.size(); ++i)  {  v[row] += s[i];  if(row==0) step = 1;  else if(row==numRows-1) step = -1;  row += step;  }  string res = "";  for(int i=0; i<numRows; ++i)  {  res += v[i];  }  return res;  }  Java1  public String convert(String s, int numRows) {  if(numRows<=1)  return s;  int dis = 2 \* (numRows-1);  int len = s.length();  char[] res = new char[len];  for(int row=0, i=0; row<numRows; ++row)  {  for(int col=row; col<len; col+=dis)  {  res[i++] = s.charAt(col);  int mid = col+dis-2\*row;  if(row>0 && row<numRows-1 && mid < len)  res[i++] = s.charAt(mid);  }  }  return new String(res);  } |
| 8. 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.  **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.  C++1  int myAtoi(string str) {  long res = 0;  int sign = 1;  int i = str.find\_first\_not\_of(" ");  if(str[i]=='-' || str[i]=='+')  {  if(str[i] == '-') sign = -1;  i++;  }  while(isdigit(str[i]))  {  res = res\*10 + str[i++]-'0';  if(res\*sign <= INT\_MIN) return INT\_MIN;  if(res\*sign >= INT\_MAX) return INT\_MAX;  }  return sign \* res;  }  Java1  public int myAtoi(String str) {  if(str.isEmpty()) return 0;  int res=0, sign=1;  int base = Integer.MAX\_VALUE/10;  int i=0;  str = str.trim();  if(str.charAt(i)=='-' || str.charAt(i)=='+')  sign = (str.charAt(i++)=='+') ? 1 : -1;  for(; i<str.length() && Character.isDigit(str.charAt(i)); ++i)  {  if(res>base || (res==base && str.charAt(i)>'7'))  return (sign==1) ? Integer.MAX\_VALUE : Integer.MIN\_VALUE;  res = 10\*res + (str.charAt(i) - '0');  }  return sign \* res;  } |
| 648. Replace Words  In English, we have a concept called root, which can be followed by some other words to form another longer word - let's call this word successor. For example, the root an, followed by other, which can form another word another.  Now, given a dictionary consisting of many roots and a sentence. You need to replace all the successor in the sentence with the root forming it. If a successor has many roots can form it, replace it with the root with the shortest length.  You need to output the sentence after the replacement.  **Example 1:**  **Input:** dict = ["cat", "bat", "rat"]  sentence = "the cattle was rattled by the battery"  **Output:** "the cat was rat by the bat"  **Note:**   1. The input will only have lower-case letters. 2. 1 <= dict words number <= 1000 3. 1 <= sentence words number <= 1000 4. 1 <= root length <= 100 5. 1 <= sentence words length <= 1000   C++1  class Solution {  public:  vector<string> getWords(string& sentence)  {  stringstream ss(sentence);  string token;  vector<string> res;  while(getline(ss, token, ' '))  res.push\_back(token);  return res;  }    string getReplaceWord(unordered\_map<string, bool>& dmap, string& word, int maxS)  {  for(int i=1; i<word.length() && i<=maxS; ++i)  {  string temp = word.substr(0, i);  if(dmap[temp] == true)  return temp;  }  return word;  }    string replaceWords(vector<string>& dict, string sentence) {  unordered\_map<string, bool> dmap;  int maxS = 1;  for(string d : dict)  {  dmap[d]=true;  if(d.length()>maxS)  maxS = d.length();  }    vector<string> words = getWords(sentence);  string res = "";  if(!words.empty())  res = getReplaceWord(dmap, words[0], maxS);  for(int i=1; i<words.size(); ++i)  res += " " + getReplaceWord(dmap, words[i], maxS);  return res;  }  };  Java1  public class Solution {  public String replaceWords(List<String> dict, String sentence) {  Map<String, Boolean> dmap = new HashMap<>();  int maxS = 1;  for(String d : dict) {  dmap.put(d, true);  if(d.length() > maxS) maxS = d.length();  }    String[] words = sentence.split(" ");  StringBuilder res = new StringBuilder();  if(words.length > 0)  res.append(getReplaceWord(dmap, words[0], maxS));  for(int i=1; i<words.length; ++i)  res.append(" " + getReplaceWord(dmap, words[i], maxS));    return res.toString();  }    private String getReplaceWord(Map<String, Boolean> dmap, String word, int maxS){  for(int i=1; i<word.length() && i<=maxS; ++i){  String temp = word.substring(0, i);  if(dmap.containsKey(temp))  return temp;  }  return word;  }  } |
|  |

### 广义表(General List)

## 非线性结构

### 树(Tree)

#### Binary Tree

|  |
| --- |
| 114. 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.](https://leetcode.com/problems/flatten-binary-tree-to-linked-list/)  **Hints:**  If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.  C++1  class Solution {  private:  TreeNode\* pre = NULL;  public:  void flatten(TreeNode\* root) {  if(root == NULL) return;  flatten(root->right);  flatten(root->left);  root->right = pre;  root->left = NULL;  pre = root;  }  };  C++2  void flatten(TreeNode\* root) {  stack<TreeNode\*> sta;  while(root!=NULL || !sta.empty())  {  if(root->right != NULL)  sta.push(root->right);  if(root->left != NULL)  {  root->right = root->left;  root->left = NULL;  }  else if(root->left==NULL && !sta.empty())  {  TreeNode\* temp = sta.top();  sta.pop();  root->right = temp;  }  root = root->right;  }  }  Java1  public void flatten(TreeNode root) {  Stack<TreeNode> stk = new Stack<TreeNode>();  while(root!=null || !stk.empty()){  if(root.right != null){  stk.push(root.right);  }  if(root.left != null){  root.right = root.left;  root.left = null;  root = root.right;  continue;  }  if(root.left==null && !stk.empty()){  TreeNode temp = stk.pop();  root.right = temp;  }  root = root.right;  }  } |
| 129. 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.  C++1  class Solution {  public:  int sumTree(TreeNode\* root, int sum)  {  sum = sum\*10 + root->val;  if(root->left==NULL && root->right==NULL)  return sum;  else if(root->left==NULL)  return sumTree(root->right, sum);  else if(root->right==NULL)  return sumTree(root->left, sum);  else  return sumTree(root->right, sum) + sumTree(root->left, sum);  }  int sumNumbers(TreeNode\* root) {  if(root==NULL) return 0;  return sumTree(root, 0);  }  };  C++2  class Solution {  public:  int Helper(TreeNode\* root,int cur)  {  if(root==NULL) return 0;  cur = cur\*10 + root->val;  if(!root->left && !root->right) return cur;  return Helper(root->left, cur) + Helper(root->right, cur);  }  int sumNumbers(TreeNode\* root) {  return Helper(root, 0);  }  };  Java1  public class Solution {    private int res;    public int sumNumbers(TreeNode root) {  if(root==null) return 0;  res = 0;  countPathVal(root, 0);  return res;  }    private void countPathVal(TreeNode root, int pathVal){  int num = pathVal \* 10 + root.val;  if(root.left==null && root.right==null)  res += num;  if(root.left != null)  countPathVal(root.left, num);  if(root.right != null)  countPathVal(root.right, num);  }  } |
| 112. 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.  C++1 (3ms) O(n)  bool hasPathSum(TreeNode\* root, int sum) {  if(root==NULL) return false;  if(root->left==NULL && root->right==NULL)  {  if(sum == root->val) return true;  return false;  }  return hasPathSum(root->left, sum-root->val) || hasPathSum(root->right, sum-root->val);  }  Java1 (1ms) O(n)  public boolean hasPathSum(TreeNode root, int sum) {  if(root==null) return false;  int rem = sum-root.val;  if(rem==0 && root.left==null && root.right==null) return true;  return hasPathSum(root.left, rem) || hasPathSum(root.right, rem);  } |
| 113. 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]  ]  C++1 (26ms)  class Solution {  public:  void pathSumBacktracking(vector<vector<int>>& res, vector<int> aRow, TreeNode\* root, int sum)  {  if(root==NULL) return;  aRow.push\_back(root->val);  if(root->left==NULL && root->right==NULL)  {  if(sum-root->val==0)  res.push\_back(aRow);  }  else  {  pathSumBacktracking(res, aRow, root->left, sum-root->val);  pathSumBacktracking(res, aRow, root->right, sum-root->val);  }  }  vector<vector<int>> pathSum(TreeNode\* root, int sum) {  vector<vector<int>> res;  vector<int> aRow;  pathSumBacktracking(res, aRow, root, sum);  return res;  }  };  C++2 (9ms)  class Solution {  public:  void pathSumBacktracking(vector<vector<int>>& res, vector<int>& aRow, TreeNode\* root, int sum)  {  if(root==NULL) return;  aRow.push\_back(root->val);  sum -= root->val;  if(root->left==NULL && root->right==NULL && sum==0)  res.push\_back(aRow);  pathSumBacktracking(res, aRow, root->left, sum);  pathSumBacktracking(res, aRow, root->right, sum);  aRow.pop\_back();  }  vector<vector<int>> pathSum(TreeNode\* root, int sum) {  vector<vector<int>> res;  vector<int> aRow;  pathSumBacktracking(res, aRow, root, sum);  return res;  }  }; |
| 437. 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  C++1  int pathSumProcess(TreeNode\* root, int sum)  {  if(root == NULL)  return 0;  else  return pathSumProcess(root->left, sum - root->val) + pathSumProcess(root->right, sum - root->val) + ((root->val==sum) ? 1 : 0);  }  int pathSum(TreeNode\* root, int sum) {  if(root==NULL) return 0;  return pathSumProcess(root, sum) + pathSum(root->left, sum) + pathSum(root->right, sum);  } |
| 101. Symmetric Tree  Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).  For example, this binary tree [1,2,2,3,4,4,3] is symmetric:  1  / \  2 2  / \ / \  3 4 4 3  But the following [1,2,2,null,3,null,3] is not:  1  / \  2 2  \ \  3 3  C++1 (8ms)  bool Symmetric(TreeNode\* left, TreeNode\* right)  {  if(left==NULL || right == NULL) return left==right;  if(left->val != right->val) return false;  return Symmetric(left->left, right->right) && Symmetric(left->right, right->left);  }  bool isSymmetric(TreeNode\* root) {  return root==NULL || Symmetric(root->left, root->right);  }  C++2 (8ms)  bool isSymmetric(TreeNode\* root) {  if(!root) return true;  stack<TreeNode\*> s;  s.push(root->left);  s.push(root->right);  TreeNode \*A, \*B;  while(!s.empty())  {  A = s.top();  s.pop();  B = s.top();  s.pop();  if(!A && !B) continue;  if(!A || !B) return false;  if(A->val != B->val) return false;  s.push(A->left);  s.push(B->right);  s.push(B->left);  s.push(A->right);  }  return true;  }  Java1 (1ms)  public boolean isSymmetric(TreeNode root) {  return root==null || symmetric(root.left, root.right);  }    private boolean symmetric(TreeNode Left, TreeNode Right)  {  if(Left==null || Right==null) return Left==Right;  else if(Left.val!=Right.val) return false;  else return symmetric(Left.left, Right.right) && symmetric(Left.right, Right.left);  }  Java2 (3ms)  public boolean isSymmetric(TreeNode root) {  if(root==null) return true;  Stack<TreeNode> s = new Stack<TreeNode>();  s.push(root.left);  s.push(root.right);  while(!s.empty())  {  TreeNode child1 = s.pop();  TreeNode child2 = s.pop();  if(child1==null && child2==null) continue;  else if(child1==null || child2==null) return false;  else if(child1.val != child2.val) return false;  s.push(child1.left);  s.push(child2.right);  s.push(child1.right);  s.push(child2.left);  }  return true;  } |
| 111. 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.  C++1 (9ms)  int minDepth(TreeNode\* root) {  if(root==NULL) return 0;  if(root->left==NULL && root->right==NULL) return 1;  if(root->left!=NULL && root->right!=NULL)  return min(minDepth(root->left), minDepth(root->right))+1;  if(root->left!=NULL) return minDepth(root->left) + 1;  if(root->right!=NULL) return minDepth(root->right) + 1;  }  C++2 (12ms) O(n)  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 min(minDepth(root->left), minDepth(root->right)) + 1;  }  C++2 (12ms)  int minDepth(TreeNode\* root) {  if(!root) return 0;  queue<TreeNode\*> q;  q.push(root);  int i=0;  while(!q.empty())  {  i++;  int size = q.size();  for(int j=0; j<size; j++)  {  TreeNode\* t = q.front();  if(!t->left && !t->right) return i;  if(t->left) q.push(t->left);  if(t->right) q.push(t->right);  q.pop();  }  }  return i;  }  Java1 (2ms)  public int minDepth(TreeNode root) {  if(root==null) return 0;  else if(root.left==null && root.right==null) return 1;  else if(root.left!=null && root.right!=null)  return Math.min(minDepth(root.left), minDepth(root.right)) + 1;  else if(root.left!=null)  return minDepth(root.left) + 1;  else  return minDepth(root.right) + 1;  } |
| 337. 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**.  C++1 (16ms)  class Solution {  public:  vector<int> robBT(TreeNode\* root)  {  vector<int> curNext(2);  if(root==NULL) return curNext;  vector<int> left = robBT(root->left);  vector<int> right = robBT(root->right);  curNext[0] = root->val + left[1] + right[1]; //rob current layer  curNext[1] = max(left[0], left[1]) + max(right[0], right[1]);  return curNext;  }  int rob(TreeNode\* root) {  vector<int>res = robBT(root);  return max(res[0], res[1]);  }  };  Java1 (1ms)  public class Solution {  public int rob(TreeNode root) {  int[] rebSum = robTree(root);  return Math.max(rebSum[0], rebSum[1]);  }    private int[] robTree(TreeNode root)  {  int[] robSum = new int[2];  if(root==null) return robSum;  int[] robLeft = robTree(root.left);  int[] robRight = robTree(root.right);  robSum[0] = Math.max(robLeft[0], robLeft[1])  + Math.max(robRight[0], robRight[1]);  robSum[1] = root.val + robLeft[0] + robRight[0];  return robSum;  }  } |
| 199. 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 <---  / \  2 3 <---  \ \  5 4 <---  You should return [1, 3, 4].  C++1 (3ms) O(n) using queue  vector<int> rightSideView(TreeNode\* root) {  vector<int> res;  if(root==NULL) return res;  queue<TreeNode\*> q;  q.push(root);  while(!q.empty())  {  int size = q.size();  res.push\_back(q.front()->val);  for(int i=0; i<size; ++i)  {  TreeNode\* cur = q.front();  q.pop();  if(cur->right != NULL) q.push(cur->right);  if(cur->left != NULL) q.push(cur->left);  }  }  return res;  }  C++2  vector<int> rightSideView(TreeNode\* root) {  vector<int> res;  if(root==NULL) return res;  queue<TreeNode\*> q;  q.push(root);  TreeNode\* t;  while(!q.empty())  {  q.push(NULL);  while(q.front()!=NULL)  {  t = q.front();  q.pop();  if(q.front()==NULL)  {  res.push\_back(t->val);  }  if(t->left) q.push(t->left);  if(t->right) q.push(t->right);  }  q.pop();  }  return res;  }  C++3  vector<int> rightSideView(TreeNode\* root) {  vector<int> res;  if(root==NULL) return res;  queue<TreeNode\*> q;  q.push(root);  TreeNode\* t;  int len;  while(!q.empty())  {  len = q.size();  for(int i=0; i<len; ++i)  {  t = q.front();  q.pop();  if(t->left) q.push(t->left);  if(t->right) q.push(t->right);  }  res.push\_back(t->val);  }  return res;  }  C++4 recursive  class Solution {  public:  void helper(TreeNode\* root, vector<int>& res, int depth)  {  if(root==NULL) return;  if(res.size()==depth) res.push\_back(root->val);  helper(root->right, res, depth+1);  helper(root->left, res, depth+1);  }  vector<int> rightSideView(TreeNode\* root) {  vector<int> res;  helper(root, res, 0);  return res;  }  };  Java1  public List<Integer> rightSideView(TreeNode root) {  List<Integer> res = new LinkedList<Integer>();  if(root == null) return res;  Queue<TreeNode> q = new LinkedList<TreeNode>();  q.offer(root);  while(!q.isEmpty())  {  int qSize = q.size();  res.add(q.peek().val);  for(int i=0; i<qSize; ++i)  {  TreeNode temp = q.poll();  if(temp.right != null)  q.offer(temp.right);  if(temp.left != null)  q.offer(temp.left);  }  }  return res;  }  Java2  public class Solution {    private List<Integer> res;    public List<Integer> rightSideView(TreeNode root) {  res = new LinkedList<Integer>();  recursiveAddNodes(root, 0);  return res;  }    private void recursiveAddNodes(TreeNode root, int depth)  {  if(root == null) return;  if(res.size() == depth)  res.add(root.val);  recursiveAddNodes(root.right, depth+1);  recursiveAddNodes(root.left, depth+1);  }  } |
| 104. Maximum Depth of Binary Tree  Given a binary tree, find its maximum depth.  The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.  C++1  int maxDepth(TreeNode\* root) {  if(root == NULL) return 0;  return max(maxDepth(root->left), maxDepth(root->right)) + 1;  }  Java1  public int maxDepth(TreeNode root) {  return (root==null) ? 0 : 1 + Math.max(maxDepth(root.left), maxDepth(root.right));  }  Pythone1  def maxDepth(self, root):  if root is None:  return 0  else:  return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1 |
| 226. 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](https://twitter.com/mxcl/status/608682016205344768) by [Max Howell](https://twitter.com/mxcl):  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.  C++1 (0ms)  TreeNode\* invertTree(TreeNode\* root) {  if(root == NULL)  return root;  if(root->left == NULL && root->right == NULL)  return root;  swap(root->left, root->right);  root->left = invertTree(root->left);  root->right = invertTree(root->right);  return root;  }  C++2  TreeNode\* invertTree(TreeNode\* root) {  if(root==NULL) return root;  swap(root->left, root->right);  root->left = invertTree(root->left);  root->right = invertTree(root->right);  return root;  }  Java1 (0ms)  public TreeNode invertTree(TreeNode root) {  if(root==null) return null;  TreeNode l = invertTree(root.left);  TreeNode r = invertTree(root.right);  root.left = r;  root.right = l;  return root;  } |
| 404. 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. Return **24**.  C++1 (3ms)  int sumOfLeftLeaves(TreeNode\* root) {  if(root == NULL)  return 0;  if(root->left == NULL)  return sumOfLeftLeaves(root->right);  if(root->left->left==NULL && root->left->right==NULL)  return root->left->val + sumOfLeftLeaves(root->right);  return  sumOfLeftLeaves(root->left) + sumOfLeftLeaves(root->right);  } |
| 100. 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.  C++1 (0ms)  bool 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 false;  return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);  } |
| 331. Verify Preorder Serialization of a Binary Tree  One way to serialize a binary tree is to use pre-order 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  C++1  bool isValidSerialization(string preorder) {  int res = 1;  stringstream ss(preorder);  string node;  while(getline(ss, node, ','))  {  if(--res < 0) return false;  if(node!="#")  res += 2;  }  return res == 0;  } |
| 543. Diameter of Binary Tree  Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the **longest** path between any two nodes in a tree. This path may or may not pass through the root.  **Example:** Given a binary tree  1  / \  2 3  / \  4 5  Return **3**, which is the length of the path [4,2,1,3] or [5,2,1,3].  **Note:** The length of path between two nodes is represented by the number of edges between them.  C++1 (9ms)  class Solution {  public:  int DepthOf(TreeNode\* root, int& diameter)  {  if(root==NULL) return 0;  int depL = DepthOf(root->left, diameter);  int diameterR = 0;  int depR = DepthOf(root->right, diameterR);  diameter = max(depL+depR+1, max(diameter, diameterR));  return max(depL, depR) + 1;  }  int diameterOfBinaryTree(TreeNode\* root) {  if(root==NULL) return 0;  int res = 0;  DepthOf(root, res);  return res-1;  }  };  C++2 (9ms)  class Solution {  public:  int DepthOf(TreeNode\* root, int& diameter)  {  if(root==NULL) return 0;  int depL = DepthOf(root->left, diameter);  int depR = DepthOf(root->right, diameter);  diameter = max(depL+depR, diameter);  return max(depL, depR) + 1;  }  int diameterOfBinaryTree(TreeNode\* root) {  int res = 0;  DepthOf(root, res);  return res;  }  }; |
| 606. Construct String from Binary Tree  You need to construct a string consists of parenthesis and integers from a binary tree with the preorder traversing way.  The null node needs to be represented by empty parenthesis pair "()". And you need to omit all the empty parenthesis pairs that don't affect the one-to-one mapping relationship between the string and the original binary tree.  **Example 1:**  **Input:** Binary tree: [1,2,3,4]  1  / \  2 3  /  4  **Output:** "1(2(4))(3)"  **Explanation:** Originallay it needs to be "1(2(4)())(3()())",  but you need to omit all the unnecessary empty parenthesis pairs.  And it will be "1(2(4))(3)".  **Example 2:**  **Input:** Binary tree: [1,2,3,null,4]  1  / \  2 3  \  4  **Output:** "1(2()(4))(3)"  **Explanation:** Almost the same as the first example,  except we can't omit the first parenthesis pair to break the one-to-one mapping relationship between the input and the output.  C++1  string tree2str(TreeNode\* t) {  if(t==NULL) return "";  string res = to\_string(t->val);  if(t->left==NULL && t->right==NULL)  return res;  else if(t->left!=NULL && t->right==NULL)  return res + "(" + tree2str(t->left) + ")";  return res + "(" + tree2str(t->left) + ")" + "(" + tree2str(t->right) + ")";  } |
| 617. Merge Two Binary Trees  Given two binary trees and imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not.  You need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of new tree.  **Example 1:**  **Input:**  Tree 1 Tree 2  1 2  / \ / \  3 2 1 3  / \ \  5 4 7  **Output:**  Merged tree:  3  / \  4 5  / \ \  5 4 7  **Note:** The merging process must start from the root nodes of both trees.  C++1  TreeNode\* mergeTrees(TreeNode\* t1, TreeNode\* t2) {  if(t1==NULL && t2==NULL) return NULL;  else if(t2==NULL) return t1;  else if(t1==NULL) return t2;  t1->val += t2->val;  t1->left = mergeTrees(t1->left, t2->left);  t1->right = mergeTrees(t1->right, t2->right);  return t1;  } |
| 623. Add One Row to Tree  Given the root of a binary tree, then value v and depth d, you need to add a row of nodes with value v at the given depth d. The root node is at depth 1.  The adding rule is: given a positive integer depth d, for each NOT null tree nodes N in depth d-1, create two tree nodes with value v as N's left subtree root and right subtree root. And N's **original left subtree** should be the left subtree of the new left subtree root, its **original right subtree** should be the right subtree of the new right subtree root. If depth d is 1 that means there is no depth d-1 at all, then create a tree node with value **v** as the new root of the whole original tree, and the original tree is the new root's left subtree.  **Example 1:**  **Input:**  A binary tree as following:  4  / \  2 6  / \ /  3 1 5  **v = 1**  **d = 2**  **Output:**  4  / \  1 1  / \  2 6  / \ /  3 1 5  **Example 2:**  **Input:**  A binary tree as following:  4  /  2  / \  3 1  **v = 1**  **d = 3**  **Output:**  4  /  2  / \  1 1  / \  3 1  **Note:**   1. The given d is in range [1, maximum depth of the given tree + 1]. 2. The given binary tree has at least one tree node.   C++1  TreeNode\* addOneRow(TreeNode\* root, int v, int d) {  if(d==1)  {  TreeNode\* newRoot = new TreeNode(v);  newRoot->left = root;  return newRoot;  }  else if(d==2)  {  TreeNode\* ln = new TreeNode(v);  TreeNode\* rn = new TreeNode(v);;  ln->left = root->left;  rn->right = root->right;  root->left = ln;  root->right = rn;  return root;  }  if(root->left != NULL)  root->left = addOneRow(root->left, v, d-1);  if(root->right != NULL)  root->right = addOneRow(root->right, v, d-1);  return root;  } |
|  |

#### Binary Search Tree (BST)

|  |
| --- |
| 108. Convert Sorted Array to Binary Search Tree  Given an array where elements are sorted in ascending order, convert it to a height balanced BST.  C++1 (12ms)  class Solution {  public:  TreeNode\* getNodeFrom(vector<int>& nums, int i, int j)  {  if(i>=j) return NULL;  if(i==j-1) return new TreeNode(nums[i]);  int mid = i + (j-i)/2;  TreeNode\* root = new TreeNode(nums[mid]);  root->left = getNodeFrom(nums, i, mid);  root->right = getNodeFrom(nums, mid+1, j);  return root;  }  TreeNode\* sortedArrayToBST(vector<int>& nums) {  return getNodeFrom(nums, 0, nums.size());  }  };  Java1 (1ms)  public class Solution {  public TreeNode sortedArrayToBST(int[] nums) {  return ArrayToBST(nums, 0, nums.length-1);  }    private TreeNode ArrayToBST(int[] nums, int start, int end){  if(start>end) return null;  if(start==end)  return new TreeNode(nums[end]);  int mid = (start+end)/2;  TreeNode node = new TreeNode(nums[mid]);  node.left = ArrayToBST(nums, start, mid-1);  node.right = ArrayToBST(nums, mid+1, end);  return node;  }  } |
| 109. 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.  C++1  class Solution {  public:  TreeNode\* getBST(ListNode\* head, ListNode\* tail)  {  if(head==tail) return NULL;  ListNode\* slow = head;  ListNode\* fast = head;  while(fast!=tail && fast->next!=tail)  {  slow=slow->next;  fast = fast->next->next;  }  TreeNode\* root = new TreeNode(slow->val);  root->left = getBST(head, slow);  root->right = getBST(slow->next, tail);  return root;  }  TreeNode\* sortedListToBST(ListNode\* head) {  return getBST(head, NULL);  }  }; |
| 538. Convert BST to Greater Tree  Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus sum of all keys greater than the original key in BST.  **Example:**  **Input:** The root of a Binary Search Tree like this:  5  / \  2 13  **Output:** The root of a Greater Tree like this:  18  / \  20 13  C++1 (39ms)  class Solution {  public:  int sum;  void SumBST(TreeNode\* root)  {  if(root==NULL) return;  SumBST(root->right);  sum += root->val;  root->val = sum;  SumBST(root->left);  }  TreeNode\* convertBST(TreeNode\* root) {  sum=0;  SumBST(root);  return root;  }  }; |
| 96. 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 C++1 **Time Limit Exceeded** int numTrees(int n) {  if(n<=1) return 1;  int res = 0;  for(int i=1; i<=n; ++i)  {  int left = numTrees(i-1);  int right = numTrees(n-i);  res += left\*right;  }  return res;  }  C++2 (0ms) O(n2) O(n)space  int numTrees(int n) {  vector<int> dp(n+1);  dp[0] = dp[1] = 1;  for(int i=2; i<=n; ++i)  for(int j=0; j<i; ++j)  dp[i] += dp[j] \* dp[i-j-1];  return dp[n];  } |
| 173. 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.  /\*\*  \* Your BSTIterator will be called like this:  \* BSTIterator i = BSTIterator(root);  \* while (i.hasNext()) cout << i.next();  \*/  C++1 (19ms) using stack  class BSTIterator {  public:  stack<TreeNode\*> left;  void PushLeftToStack(TreeNode\* root)  {  while(root!=NULL)  {  left.push(root);  root=root->left;  }  }  BSTIterator(TreeNode \*root) {  PushLeftToStack(root);  }  /\*\* @return whether we have a next smallest number \*/  bool hasNext() {  return !left.empty();  }  /\*\* @return the next smallest number \*/  int next() {  TreeNode\* cur = left.top();  left.pop();  int res = cur->val;    cur = cur->right;  PushLeftToStack(cur);  return res;  }  };  C++2  class BSTIterator {  private:  stack<TreeNode\*> myStack;  void pushLeft(TreeNode\* root)  {  for(;root!=NULL; myStack.push(root), root = root->left);  }  public:  BSTIterator(TreeNode \*root) {  pushLeft(root);  }  /\*\* @return whether we have a next smallest number \*/  bool hasNext() {  return !myStack.empty();  }  /\*\* @return the next smallest number \*/  int next() {  TreeNode\* t = myStack.top();  myStack.pop();  pushLeft(t->right);  return t->val;  }  };  Java1  public class BSTIterator {  private Stack<TreeNode> smallList;  public BSTIterator(TreeNode root) {  smallList = new Stack<TreeNode>();  storeLeft(root);  }    private void storeLeft(TreeNode root)  {  while(root != null)  {  smallList.push(root);  root = root.left;  }  }  /\*\* @return whether we have a next smallest number \*/  public boolean hasNext() {  return !smallList.empty();  }  /\*\* @return the next smallest number \*/  public int next() {  TreeNode cur = smallList.pop();  storeLeft(cur.right);  return cur.val;  }  } |
| 449. Serialize and Deserialize BST  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 search tree**. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary search tree can be serialized to a string and this string can be deserialized to the original tree structure.  **The encoded string should be as compact as possible.**  **Note:** Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.  C++1 (36ms)  class Codec {  public:  void getStr(TreeNode\* root, string& s)  {  if(root==NULL) return;  s+= ","+to\_string(root->val);  getStr(root->left, s);  getStr(root->right, s);  }  // Encodes a tree to a single string.  string serialize(TreeNode\* root) {  if(root==NULL) return "";  string res;  getStr(root, res);  return res.substr(1);  }  vector<int> getNodesVal(string data)  {  stringstream ss(data);  string token;  vector<int> res;  while(getline(ss, token, ','))  res.push\_back(stoi(token));  return res;  }  TreeNode\* getNodes(queue<int>& q)  {  if(q.empty()) return NULL;  TreeNode\* root = new TreeNode(q.front());  q.pop();  queue<int> subQ;  while(!q.empty() && q.front() < root->val)  {  subQ.push(q.front());  q.pop();  }  root->left = getNodes(subQ);  root->right = getNodes(q);  return root;  }  // Decodes your encoded data to tree.  TreeNode\* deserialize(string data) {  vector<int> nodes = getNodesVal(data);  queue<int> q;  for(int n : nodes)  q.push(n);  return getNodes(q);  }  };  C++2 (46ms)  class Codec {  public:  string s;  // Encodes a tree to a single string.  string serialize(TreeNode\* root) {  if(root==NULL) return "";  s += ","+to\_string(root->val);  serialize(root->left);  serialize(root->right);  return s.substr(1);  }  vector<int> getNodesVal(string data)  {  stringstream ss(data);  string token;  vector<int> res;  while(getline(ss, token, ','))  res.push\_back(stoi(token));  return res;  }  TreeNode\* getNodes(queue<int>& q)  {  if(q.empty()) return NULL;  TreeNode\* root = new TreeNode(q.front());  q.pop();  queue<int> subQ;  while(!q.empty() && q.front() < root->val)  {  subQ.push(q.front());  q.pop();  }  root->left = getNodes(subQ);  root->right = getNodes(q);  return root;  }  // Decodes your encoded data to tree.  TreeNode\* deserialize(string data) {  vector<int> nodes = getNodesVal(data);  queue<int> q;  for(int n : nodes)  q.push(n);  return getNodes(q);  }  }; |
| 95. Unique Binary Search Trees II  Given an integer *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  C++1  class Solution {  public:  vector<TreeNode\*> generateBST(int start, int end)  {  vector<TreeNode\*> res;  if(start > end)  {  res.push\_back(NULL);  return res;  }  else if(start == end)  {  TreeNode\* leaf = new TreeNode(start);  res.push\_back(leaf);  return res;  }  for(int i=start; i<=end; ++i)  {  vector<TreeNode\*> leftNodes = generateBST(start, i-1);  vector<TreeNode\*> rightNodes = generateBST(i+1, end);  for(auto left : leftNodes)  {  for(auto right : rightNodes)  {  TreeNode\* root = new TreeNode(i);  root->left = left;  root->right = right;  res.push\_back(root);  }  }  }  return res;  }  vector<TreeNode\*> generateTrees(int n) {  if(n<1) return vector<TreeNode\*>();  return generateBST(1, n);  }  }; |
| 98. 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.   **Example 1:**  2  / \  1 3  Binary tree [2,1,3], return true.  **Example 2:**  1  / \  2 3  Binary tree [1,2,3], return false.  C++1  class Solution {  public:  bool isBST(TreeNode\* root, TreeNode\* small, TreeNode\* great)  {  if(root==NULL) return true;  if((small!=NULL && small->val >= root->val) || (great!=NULL && root->val >=great->val)) return false;  return isBST(root->left, small, root) && isBST(root->right, root, great);  }  bool isValidBST(TreeNode\* root) {  return isBST(root, NULL, NULL);  }  }; |
| 669. Trim a Binary Search Tree  Given a binary search tree and the lowest and highest boundaries as L and R, trim the tree so that all its elements lies in [L, R] (R >= L). You might need to change the root of the tree, so the result should return the new root of the trimmed binary search tree.  **Example 1:**  **Input:**  1  / \  0 2  L = 1  R = 2  **Output:**  1  \  2  **Example 2:**  **Input:**  3  / \  0 4  \  2  /  1  L = 1  R = 3  **Output:**  3  /  2  /  1  C++1  TreeNode\* trimBST(TreeNode\* root, int L, int R) {  if(root == NULL) return NULL;  if(root->val < L)  {  root = root->right;  return trimBST(root, L, R);  }  if(root->val > R)  {  root = root->left;  return trimBST(root, L, R);  }  if(root->val == L) root->left = NULL;  if(root->val == R) root->right = NULL;  root->left = trimBST(root->left, L, R);  root->right = trimBST(root->right, L, R);  return root;  } |
|  |

#### Balanced Binary Tree

|  |
| --- |
| 110. 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.  C++1 (16ms)  int depthOfTree(TreeNode\* root)  {  if(root==NULL) return 0;  return max(depthOfTree(root->left),depthOfTree(root->right)) + 1;  }  bool isBalanced(TreeNode\* root) {  if(root==NULL) return true;  if(abs(depthOfTree(root->left) - depthOfTree(root->right)) > 1) return false;  return isBalanced(root->left) && isBalanced(root->right);  }  C++2 (16ms)  int MaxTreeDepth(TreeNode\* root)  {  if(!root) return 0;  return max(MaxTreeDepth(root->left), MaxTreeDepth(root->right)) + 1;  }  bool isBalanced(TreeNode\* root) {  if(!root) return true;  int LDepth = MaxTreeDepth(root->left);  int RDepth = MaxTreeDepth(root->right);  return abs(LDepth - RDepth)<=1 && isBalanced(root->left) && isBalanced(root->right);  }  Java1 (2ms)  public boolean isBalanced(TreeNode root) {  if(root==null) return true;  else if(Math.abs(getDepth(root.left)-getDepth(root.right)) > 1) return false;  else  return isBalanced(root.left) && isBalanced(root.right);  }  public int getDepth(TreeNode root)  {  if(root==null) return 0;  else  return Math.max(getDepth(root.left),getDepth(root.right)) + 1;  } |
|  |
|  |

#### Complete Tree

|  |
| --- |
| 222. Count Complete Tree Nodes  Given a **complete** binary tree, count the number of nodes.  **Definition of a complete binary tree from**[**Wikipedia**](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees)**:** 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 2h nodes inclusive at the last level h.  C++1 (TLE)  int countNodes(TreeNode\* root) {  if(root==NULL) return 0;  return 1 + countNodes(root->left) + countNodes(root->right);  }  C++2  int countNodes(TreeNode\* root) {  if(root==NULL) return 0;  TreeNode \*lt=root, \*rt=root;  int lh=0, rh=0;  while(lt)  {  lh++;  lt=lt->left;  }  while(rt)  {  rh++;  rt = rt->right;  }  if(lh==rh) return pow(2,lh)-1;  return 1 + countNodes(root->left) + countNodes(root->right);  }  C++3  int countNodes(TreeNode\* root) {  if(root==NULL) return 0;  TreeNode \*L=root, \*R=root;  int height=0;  while(R!=NULL)  {  L = L->left;  R = R->right;  height++;  }  if(L==NULL) return (1<<height) - 1;  return 1 + countNodes(root->left) + countNodes(root->right);  }  C++4  class Solution {  public:  int height(TreeNode\* root)  {  return root==NULL ? -1 : 1+height(root->left);  }  int countNodes(TreeNode\* root) {  if(root==NULL) return 0;  int h = height(root);  return h-1==height(root->right) ? (1<<h) + countNodes(root->right) //left is full tree  : (1<<(h-1)) + countNodes(root->left); //right is full tree  }  };  C++5  class Solution {  public:  int height(TreeNode\* root)  {  return root==NULL ? -1 : 1+height(root->left);  }  int countNodes(TreeNode\* root) {  int h = height(root), nodes=0;  while(root!=NULL)  {  if(h-1 == height(root->right))  {  nodes+= (1<<h);  root=root->right;  }  else  {  nodes += (1<<(h-1));  root = root->left;  }  h--;  }  return nodes;  }  }; |
|  |
|  |
|  |

#### Traversal

|  |
| --- |
| 103. 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,null,null,15,7],  3  / \  9 20  / \  15 7  return its zigzag level order traversal as:  [  [3],  [20,9],  [15,7]  ]  C++1  vector<vector<int>> zigzagLevelOrder(TreeNode\* root) {  vector<vector<int>> res;  stack<TreeNode\*> sta;  if(root==NULL) return res;  bool LToR = true;  sta.push(root);  while(!sta.empty())  {  int size = sta.size();  vector<int> aRow;  stack<TreeNode\*> subS;  for(int i=0; i<size; ++i)  {  TreeNode\* temp = sta.top();  sta.pop();  aRow.push\_back(temp->val);  if(LToR)  {  if(temp->left!=NULL) subS.push(temp->left);  if(temp->right!=NULL) subS.push(temp->right);  }  else  {  if(temp->right!=NULL) subS.push(temp->right);  if(temp->left!=NULL) subS.push(temp->left);  }  }  sta = subS;  res.push\_back(aRow);  LToR = !LToR;  }  return res;  } |
| 102. 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,null,null,15,7],  3  / \  9 20  / \  15 7  return its level order traversal as:  [  [3],  [9,20],  [15,7]  ]  C++1 (9ms)  vector<vector<int>> levelOrder(TreeNode\* root) {  vector<vector<int>> res;  if(root==NULL) return res;  queue<TreeNode\*> q({root});  while(!q.empty())  {  int size = q.size();  vector<int> aRow;  for(int i=0; i<size; ++i)  {  TreeNode\* cur = q.front();  q.pop();  aRow.push\_back(cur->val);  if(cur->left!=NULL) q.push(cur->left);  if(cur->right!=NULL) q.push(cur->right);  }  res.push\_back(aRow);  }  return res;  }  C++2  class Solution {  public:  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;  }  };  Java1 (3ms)  public List<List<Integer>> levelOrder(TreeNode root) {  List<List<Integer>> res = new ArrayList<>();  if(root==null) return res;  LinkedList<TreeNode> q1 = new LinkedList<TreeNode>();  q1.add(root);  while(!q1.isEmpty())  {  LinkedList<TreeNode> q2 = new LinkedList<TreeNode>();  List<Integer> arow = new ArrayList<>();  while(!q1.isEmpty())  {  TreeNode node = q1.poll();  arow.add(node.val);  if(node.left!=null) q2.add(node.left);  if(node.right!=null) q2.add(node.right);  }  res.add(arow);  q1=q2;  }  return res;  }  Java2 (3ms)  public List<List<Integer>> levelOrder(TreeNode root) {  List<List<Integer>> res = new LinkedList<>();  if(root==null) return res;  Queue<TreeNode> q1 = new LinkedList<TreeNode>();  q1.offer(root);  while(!q1.isEmpty())  {  Queue<TreeNode> q2 = new LinkedList<TreeNode>();  List<Integer> arow = new LinkedList<>();  while(!q1.isEmpty())  {  TreeNode node = q1.poll();  arow.add(node.val);  if(node.left!=null) q2.offer(node.left);  if(node.right!=null) q2.offer(node.right);  }  res.add(arow);  q1=q2;  }  return res;  } |
| 107. 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,null,null,15,7],  3  / \  9 20  / \  15 7  return its bottom-up level order traversal as:  [  [15,7],  [9,20],  [3]  ]  C++1 (3ms, beats 94.13% )  class Solution {  private:  int depth;  public:  int GetDepth(TreeNode\* root)  {  if(root==NULL) return 0;  return max(GetDepth(root->left), GetDepth(root->right)) + 1;  }  void WriteIntoVector(vector<vector<int>>& res, TreeNode\* root, int row)  {  if(root==NULL) return;  res[row].push\_back(root->val);  WriteIntoVector(res, root->left, row-1);  WriteIntoVector(res, root->right, row-1);  }  vector<vector<int>> levelOrderBottom(TreeNode\* root) {  depth = GetDepth(root);  vector<vector<int>> res(depth);  WriteIntoVector(res, root, depth-1);  return res;  }  };  C++2 (6ms beats 29.76%)  vector<vector<int>> levelOrderBottom(TreeNode\* root) {  vector<vector<int>> res;  if(root==NULL) return res;  queue<TreeNode\*> q;  q.push(root);  while(!q.empty())  {  int size = q.size();  vector<int> aRow;  for(int i=0; i<size; ++i)  {  TreeNode\* t = q.front();  q.pop();  aRow.push\_back(t->val);  if(t->left != NULL) q.push(t->left);  if(t->right != NULL) q.push(t->right);  }  res.insert(res.begin(), aRow);  }  return res;  } |
|  |
|  |
| 94. Binary Tree Inorder Traversal  Given a binary tree, return the *inorder* traversal of its nodes' values.  For example: Given binary tree [1,null,2,3],  1  \  2  /  3  return [1,3,2].  **Note:** Recursive solution is trivial, could you do it iteratively?  C++1 (Recursive)  class Solution {  public:  void RecursiveInorderTraversal(TreeNode\* root, vector<int>& res)  {  if(root==NULL) return;  RecursiveInorderTraversal(root->left, res);  res.push\_back(root->val);  RecursiveInorderTraversal(root->right, res);  }  vector<int> inorderTraversal(TreeNode\* root) {  vector<int> res;  RecursiveInorderTraversal(root, res);  return res;  }  };  C++2 (Iterative) Stack  vector<int> inorderTraversal(TreeNode\* root) {  vector<int> res;  stack<TreeNode\*> sta;  TreeNode\* node = root;  while(node!=NULL || !sta.empty())  {  if(node != NULL)  {  sta.push(node);  node = node->left;  }  else  {  node = sta.top();  sta.pop();  res.push\_back(node->val);  node = node->right;  }  }  return res;  }  C++3  //Morris traversal  vector<int> inorderTraversal(TreeNode\* root) {  vector<int> res;  TreeNode\* cur = root, \* prev = NULL;  while(cur)  {  if(cur->left == NULL)  {  res.push\_back(cur->val);  cur = cur->right;  }  else  {  //firn predecessor  prev = cur->left;  while(prev->right!=NULL && prev->right!=cur) prev = prev->right;  if(prev->right==NULL)  {  prev->right = cur;  cur = cur->left;  }  else  {  prev->right = NULL;  res.push\_back(cur->val);  cur = cur->right;  }  }  }  return res;  }  Java1  public class Solution {  public List<Integer> inorderTraversal(TreeNode root) {  List<Integer> res = new LinkedList<Integer>();  recursiveIT(res, root);  return res;  }    private void recursiveIT(List<Integer> res, TreeNode root)  {  if(root == null) return;  recursiveIT(res, root.left);  res.add(root.val);  recursiveIT(res, root.right);  }  }  Java2  public List<Integer> inorderTraversal(TreeNode root) {  List<Integer> res = new LinkedList<Integer>();  TreeNode cur = root;  Stack<TreeNode> stack = new Stack<TreeNode>();  while(cur!=null || !stack.empty())  {  if(cur != null)  {  stack.push(cur);  cur = cur.left;  }  else  {  cur = stack.pop();  res.add(cur.val);  cur = cur.right;  }  }  return res;  } |
| 144. 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?  C++1 (3ms) O(n)  public:  void RecursivePreorder(TreeNode\* root, vector<int>& res)  {  if(root==NULL) return;  res.push\_back(root->val);  RecursivePreorder(root->left, res);  RecursivePreorder(root->right, res);  }  vector<int> preorderTraversal(TreeNode\* root) {  vector<int> res;  RecursivePreorder(root, res);  return res;  }  };  C++2 (3ms)  vector<int> preorderTraversal(TreeNode\* root) {  vector<int> res;  stack<TreeNode\*> sta;  if(root==NULL) return res;  sta.push(root);  while(!sta.empty())  {  TreeNode\* cur = sta.top();  sta.pop();  res.push\_back(cur->val);  if(cur->right != NULL) sta.push(cur->right);  if(cur->left != NULL) sta.push(cur->left);  }  return res;  }  Java1  public class Solution {  public List<Integer> preorderTraversal(TreeNode root) {  List<Integer> res = new ArrayList<Integer>();  recursivePT(res, root);  return res;  }  private void recursivePT(List<Integer> res, TreeNode root)  {  if(root == null) return;  res.add(root.val);  recursivePT(res, root.left);  recursivePT(res, root.right);  }  }  Java2  public List<Integer> preorderTraversal(TreeNode root) {  List<Integer> res = new LinkedList<Integer>();  if(root == null) return res;  Stack<TreeNode> stack = new Stack<TreeNode>();  stack.push(root);  while(!stack.empty())  {  TreeNode temp = stack.pop();  res.add(temp.val);  if(temp.right != null)  stack.push(temp.right);  if(temp.left != null)  stack.push(temp.left);  }  return res;  } |
| 106. 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.  C++1  class Solution {  public:  TreeNode\* BuildTreeFrom(unordered\_map<int, int>& inorderMap, vector<int>& postorder, int Pstart, int Pend, int Istart, int Iend)  {  if(Pstart>Pend) return NULL;  TreeNode\* root = new TreeNode(postorder[Pend]);  if(Pstart==Pend) return root;  int rootP = inorderMap[root->val];  int leftS = rootP-Istart;  int rightS = Iend-rootP;  root->left = BuildTreeFrom(inorderMap, postorder, Pstart, Pstart+leftS-1, Istart, rootP-1);  root->right = BuildTreeFrom(inorderMap, postorder, Pend-rightS, Pend-1, rootP+1, Iend);  return root;  }  TreeNode\* buildTree(vector<int>& inorder, vector<int>& postorder) {  unordered\_map<int, int> inorderMap;  for(int i=0; i<inorder.size(); ++i)  inorderMap[inorder[i]] = i;  return BuildTreeFrom(inorderMap, postorder, 0, postorder.size()-1, 0, inorder.size()-1);  }  }; |
| 105. 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.  C++1  class Solution {  public:  TreeNode\* BuildTreeFrom(unordered\_map<int, int>& inorderMap, vector<int>& preorder, int Pstart, int Pend, int Istart, int Iend)  {  if(Pstart>Pend) return NULL;  TreeNode\* root = new TreeNode(preorder[Pstart]);  if(Pstart==Pend) return root;  int rootP = inorderMap[root->val];  int leftS = rootP-Istart;  int rightS = Iend-rootP;  root->left = BuildTreeFrom(inorderMap, preorder, Pstart+1, Pstart+leftS, Istart, rootP-1);  root->right = BuildTreeFrom(inorderMap, preorder, Pend-rightS+1 , Pend, rootP+1, Iend);  return root;  }  TreeNode\* buildTree(vector<int>& preorder, vector<int>& inorder) {  unordered\_map<int, int> inorderMap;  for(int i=0; i<inorder.size(); ++i)  inorderMap[inorder[i]] = i;  return BuildTreeFrom(inorderMap, preorder, 0, preorder.size()-1, 0, inorder.size()-1);  }  }; |
|  |
|  |

#### Top to Down

#### Down to Top

|  |
| --- |
| 563. Binary Tree Tilt  Given a binary tree, return the tilt of the **whole tree**.  The tilt of a **tree node** is defined as the **absolute difference** between the sum of all left subtree node values and the sum of all right subtree node values. Null node has tilt 0.  The tilt of the **whole tree** is defined as the sum of all nodes' tilt.  **Example:**  **Input:**  1  / \  2 3  **Output:** 1  **Explanation:**  Tilt of node 2 : 0  Tilt of node 3 : 0  Tilt of node 1 : |2-3| = 1  Tilt of binary tree : 0 + 0 + 1 = 1  **Note:**   1. The sum of node values in any subtree won't exceed the range of 32-bit integer. 2. All the tilt values won't exceed the range of 32-bit integer.   C++1  class Solution {  public:  int SumTree(TreeNode\* root, int& tilts)  {  if(root==NULL) return 0;  int sumL = SumTree(root->left, tilts);  int sumR = SumTree(root->right, tilts);  tilts += abs(sumL - sumR);  return root->val + sumL + sumR;  }  int findTilt(TreeNode\* root) {  int tilts = 0;  SumTree(root, tilts);  return tilts;  }  }; |
|  |
|  |

#### Normal tree

|  |
| --- |
| 582. Kill Process  Given **n** processes, each process has a unique **PID (process id)** and its **PPID (parent process id)**.  Each process only has one parent process, but may have one or more children processes. This is just like a tree structure. Only one process has PPID that is 0, which means this process has no parent process. All the PIDs will be distinct positive integers.  We use two list of integers to represent a list of processes, where the first list contains PID for each process and the second list contains the corresponding PPID.  Now given the two lists, and a PID representing a process you want to kill, return a list of PIDs of processes that will be killed in the end. You should assume that when a process is killed, all its children processes will be killed. No order is required for the final answer.  **Example 1:**  **Input:**  pid = [1, 3, 10, 5]  ppid = [3, 0, 5, 3]  kill = 5  **Output:** [5,10]  **Explanation:**  3  / \  1 5  /  10  Kill 5 will also kill 10.  **Note:**   1. The given kill id is guaranteed to be one of the given PIDs. 2. n >= 1.   C++1  vector<int> killProcess(vector<int>& pid, vector<int>& ppid, int kill) {  unordered\_map<int, unordered\_set<int>> tree;  for(int i=0; i<pid.size(); ++i)  tree[ppid[i]].insert(pid[i]);  vector<int> res;  queue<int> q; //or stack  q.push(kill);  while(!q.empty())  {  int parent = q.front();  q.pop();  for(int son : tree[parent])  q.push(son);  res.push\_back(parent);  }  return res;  } |
|  |
|  |

### 堆(Heap)

### 图(Graph)

|  |
| --- |
|  |
|  |
|  |

#### 无向连通图

|  |
| --- |
| 310. 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]  **Note**:  (1) According to the [definition of tree on Wikipedia](https://en.wikipedia.org/wiki/Tree_(graph_theory)): “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.  C++1  vector<int> findMinHeightTrees(int n, vector<pair<int, int>>& edges) {  vector<unordered\_set<int>> tree(n);  vector<int> countEdges(n);  vector<int> leaves;  for(auto edge : edges)  {  tree[edge.first].insert(edge.second);  tree[edge.second].insert(edge.first);  countEdges[edge.first]++;  countEdges[edge.second]++;  }  for(int i=0; i<n; ++i)  if(countEdges[i] <= 1) //collect leaves. If only one node, count==0  leaves.push\_back(i);  while(n > 2) //result is one or two nodes, cause there is no circle  {  n -= leaves.size(); //remove leaves num from totle node num  vector<int> temp;  for(int leaf : leaves)  {  for(int node : tree[leaf])  {  tree[node].erase(leaf); //remove leaves from node  if(tree[node].size()==1) temp.push\_back(node); //node is a leaf  }  }  leaves = temp;  }  return leaves;  } |
|  |
|  |
|  |

#### 有向联通图

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 332. 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 (another trick is which airport to finish all airports) 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 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.  C++1  vector<string> findItinerary(vector<pair<string, string>> tickets) {  unordered\_map<string, multiset<string>> graph;  vector<string> res;  stack<string> sta;  sta.push("JFK");  for(auto ticket : tickets)  graph[ticket.first].insert(ticket.second);  while(!sta.empty())  {  string airport = sta.top();  if(graph[airport].size()==0)  {  res.push\_back(airport);  sta.pop();  }  else  {  auto minCity = graph[airport].begin();  graph[airport].erase(minCity);  sta.push(\*minCity);  }  }  reverse(res.begin(), res.end());  return res;  }  Testcase:   |  |  |  | | --- | --- | --- | | Id | Testcase | Result | | 1 | **[["JFK","KUL"],["JFK","NRT"],["NRT","JFK"]]** | **["JFK","NRT","JFK","KUL"]** | |  |  |  | |
| 133. 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  / \  \\_/  C++1  class Solution {  private:  unordered\_map<UndirectedGraphNode\*, UndirectedGraphNode\*> graph;  public:  UndirectedGraphNode \*cloneGraph(UndirectedGraphNode \*node) {  if(node == NULL) return NULL;  if(graph.find(node) == graph.end())  {  graph[node] = new UndirectedGraphNode(node->label);  for(auto n : node->neighbors)  {  graph[node]->neighbors.push\_back(cloneGraph(n));  }  }  return graph[node];  }  };  C++2  UndirectedGraphNode \*cloneGraph(UndirectedGraphNode \*node) {  if(node==NULL) return NULL;  unordered\_map<UndirectedGraphNode\*, UndirectedGraphNode\*> graph;  stack<UndirectedGraphNode\*> sta;  sta.push(node);  graph[node] = new UndirectedGraphNode(node->label);  while(!sta.empty())  {  UndirectedGraphNode \* cur = sta.top();  sta.pop();  for(auto neighbor : cur->neighbors)  {  if(graph.find(neighbor) == graph.end())  {  sta.push(neighbor);  graph[neighbor] = new UndirectedGraphNode(neighbor->label);  }  graph[cur]->neighbors.push\_back(graph[neighbor]);  }  }  return graph[node];  } |
| 565. Array Nesting  A zero-indexed array A consisting of N different integers is given. The array contains all integers in the range [0, N - 1].  Sets S[K] for 0 <= K < N are defined as follows:  S[K] = { A[K], A[A[K]], A[A[A[K]]], ... }.  Sets S[K] are finite for each K and should NOT contain duplicates.  Write a function that given an array A consisting of N integers, return the size of the largest set S[K] for this array.  **Example 1:**  **Input:** A = [5,4,0,3,1,6,2]  **Output:** 4  **Explanation:**  A[0] = 5, A[1] = 4, A[2] = 0, A[3] = 3, A[4] = 1, A[5] = 6, A[6] = 2.  One of the longest S[K]:  S[0] = {A[0], A[5], A[6], A[2]} = {5, 6, 2, 0}  **Note:**   1. N is an integer within the range [1, 20,000]. 2. The elements of A are all distinct. 3. Each element of array A is an integer within the range [0, N-1].   C++1  int arrayNesting(vector<int>& nums) {  int n = nums.size(), res = 0;  vector<bool> mem(n);  for(int i=0; i<n; ++i)  {  int j = i, count = 0;  while(mem[j]==false)  {  mem[j] = true;  count++;  j = nums[j];  }  res = max(res, count);  }  return res;  } |
|  |

### Set

|  |
| --- |
| 653. Two Sum IV - Input is a BST  Given a Binary Search Tree and a target number, return true if there exist two elements in the BST such that their sum is equal to the given target.  **Example 1:**  **Input:**  5  / \  3 6  / \ \  2 4 7  Target = 9  **Output:** True  **Example 2:**  **Input:**  5  / \  3 6  / \ \  2 4 7  Target = 28  **Output:** False  C++1  bool findTargetIn(unordered\_set<int>& set, TreeNode\* root, int k)  {  if(root == NULL) return false;  if(set.find(k - root->val) != set.end()) return true;  set.insert(root->val);  return findTargetIn(set, root->left, k) || findTargetIn(set, root->right, k);  }    bool findTarget(TreeNode\* root, int k) {  unordered\_set<int> set;  return findTargetIn(set, root, k);  }  Java1  public boolean findTarget(TreeNode root, int k) {  Set<Integer> set = new HashSet<>();  return findTargetIn(set, root, k);  }    private boolean findTargetIn(Set<Integer> set, TreeNode root, int k){  if(root == null) return false;  if(set.contains(k - root.val)) return true;  set.add(root.val);  return findTargetIn(set, root.left, k) || findTargetIn(set, root.right, k);  } |
|  |
|  |

# 算法(Algorithm**)**

## 模拟算法(Simulated)

## 搜索算法

### 枚举搜索(Enumeration)

### 深度优先(Depth First Search)

|  |
| --- |
| 529. Minesweeper  Let's play the minesweeper game ([Wikipedia](https://en.wikipedia.org/wiki/Minesweeper_(video_game)), [online game](http://minesweeperonline.com/))!  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:**  https://leetcode.com/static/images/problemset/minesweeper_example_1.png  **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:**  https://leetcode.com/static/images/problemset/minesweeper_example_2.png  **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.   C++1 (6ms)  class Solution {  public:  int CountMines(vector<vector<char>>& board, int x, int y, int row, int col)  {  int mines = 0;  for(int i=(x>0 ? x-1 : 0); i<= (x+1<row ? x+1 : row-1); ++i)  {  for(int j=(y>0 ? y-1 : 0); j<= (y+1<col ? y+1 : col-1); ++j)  {  if(board[i][j] == 'M')  mines++;  }  }  return mines;  }  void DFS(vector<vector<char>>& board, int x, int y, int row, int col)  {  if(board[x][y] == 'E')  {  board[x][y] = 'B';  int mines = CountMines(board,x,y,row,col);  if(mines>0) board[x][y] = '0'+mines;  if(board[x][y] == 'B')  {  for(int i=(x>0 ? x-1 : 0); i<= (x+1<row ? x+1 : row-1); ++i)  {  for(int j=(y>0 ? y-1 : 0); j<= (y+1<col ? y+1 : col-1); ++j)  {  if(board[i][j] == 'E')  DFS(board,i,j,row,col);  }  }  }  }  }  vector<vector<char>> updateBoard(vector<vector<char>>& board, vector<int>& click) {  int x=click[0], y=click[1];  if(board[x][y] == 'M')  {  board[x][y] = 'X';  return board;  }  DFS(board, x, y, board.size(), board[0].size());  return board;  }  }; |
| 200. 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  C++1  class Solution {  public:  void SearchDFS(vector<vector<char>>& grid, vector<vector<bool>>& visited, int i, int j, int row, int col)  {  visited[i][j]=true;  if(i-1>=0 && grid[i-1][j]=='1' && !visited[i-1][j])  SearchDFS(grid, visited, i-1, j, row, col);  if(i+1<row && grid[i+1][j]=='1' && !visited[i+1][j])  SearchDFS(grid, visited, i+1, j, row, col);  if(j-1>=0 && grid[i][j-1]=='1' && !visited[i][j-1])  SearchDFS(grid, visited, i, j-1, row, col);  if(j+1<col && grid[i][j+1]=='1' && !visited[i][j+1])  SearchDFS(grid, visited, i, j+1, row, col);  }  int numIslands(vector<vector<char>>& grid) {  if(grid.empty()) return 0;  int row = grid.size();  int col = grid[0].size();  vector<vector<bool>> visited(row, vector<bool>(col, false));  int counts=0;  for(int i=0; i<row; ++i)  {  for(int j=0; j<col; ++j)  {  if(grid[i][j]=='1' && !visited[i][j])  {  counts++;  SearchDFS(grid, visited, i, j, row, col);  }  }  }  return counts;  }  };  C++2  class Solution {  public:  vector<int> edges = {-1, 1};  void dfs(vector<vector<char>>& grid, vector<vector<bool>>& visited, int i, int j)  {  visited[i][j] = true;  for(int e : edges)  {  if(i+e>=0 && i+e<grid.size() && grid[i+e][j]=='1' && !visited[i+e][j])  dfs(grid, visited, i+e, j);  if(j+e>=0 && j+e<grid[0].size() && grid[i][j+e]=='1' && !visited[i][j+e])  dfs(grid, visited, i, j+e);  }  }  int numIslands(vector<vector<char>>& grid) {  int h=grid.size();  int w = (h>0) ? grid[0].size() : 0;  int res = 0;  vector<vector<bool>> visited(h, vector<bool>(w));  for(int i=0; i<h; ++i)  {  for(int j=0; j<w; ++j)  {  if(grid[i][j]=='1' && !visited[i][j])  {  res++;  dfs(grid, visited, i, j);  }  }  }  return res;  }  };  C++3  class Solution {  public:  vector<int> edges = {-1, 1};  void dfs(vector<vector<char>>& grid, int i, int j)  {  grid[i][j] = '0';  for(int e : edges)  {  if(i+e>=0 && i+e<grid.size() && grid[i+e][j]=='1')  dfs(grid, i+e, j);  if(j+e>=0 && j+e<grid[0].size() && grid[i][j+e]=='1')  dfs(grid, i, j+e);  }  }  int numIslands(vector<vector<char>>& grid) {  int h=grid.size();  int w = (h>0) ? grid[0].size() : 0;  int res = 0;  for(int i=0; i<h; ++i)  {  for(int j=0; j<w; ++j)  {  if(grid[i][j]=='1')  {  res++;  dfs(grid, i, j);  }  }  }  return res;  }  }; |
| 568. Maximum Vacation Days  LeetCode wants to give one of its best employees the option to travel among **N** cities to collect algorithm problems. But all work and no play makes Jack a dull boy, you could take vacations in some particular cities and weeks. Your job is to schedule the traveling to maximize the number of vacation days you could take, but there are certain rules and restrictions you need to follow.  **Rules and restrictions:**   1. You can only travel among **N** cities, represented by indexes from 0 to N-1. Initially, you are in the city indexed 0 on **Monday**. 2. The cities are connected by flights. The flights are represented as a **N\*N** matrix (not necessary symmetrical), called **flights** representing the airline status from the city i to the city j. If there is no flight from the city i to the city j, **flights[i][j] = 0**; Otherwise, **flights[i][j] = 1**. Also, **flights[i][i] = 0** for all i. 3. You totally have **K** weeks (**each week has 7 days**) to travel. You can only take flights at most once **per day** and can only take flights on each week's **Monday** morning. Since flight time is so short, we don't consider the impact of flight time. 4. For each city, you can only have restricted vacation days in different weeks, given an **N\*K** matrix called **days** representing this relationship. For the value of **days[i][j]**, it represents the maximum days you could take vacation in the city **i** in the week **j**.   You're given the **flights** matrix and **days** matrix, and you need to output the maximum vacation days you could take during **K** weeks.  **Example 1:**  **Input:**flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[1,3,1],[6,0,3],[3,3,3]]  **Output:** 12  **Explanation:**  Ans = 6 + 3 + 3 = 12.  One of the best strategies is:  1st week : fly from city 0 to city 1 on Monday, and play 6 days and work 1 day.  (Although you start at city 0, we could also fly to and start at other cities since it is Monday.)  2nd week : fly from city 1 to city 2 on Monday, and play 3 days and work 4 days.  3rd week : stay at city 2, and play 3 days and work 4 days.  **Example 2:**  **Input:**flights = [[0,0,0],[0,0,0],[0,0,0]], days = [[1,1,1],[7,7,7],[7,7,7]]  **Output:** 3  **Explanation:**  Ans = 1 + 1 + 1 = 3.  Since there is no flights enable you to move to another city, you have to stay at city 0 for the whole 3 weeks.  For each week, you only have one day to play and six days to work.  So the maximum number of vacation days is 3.  **Example 3:**  **Input:**flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[7,0,0],[0,7,0],[0,0,7]]  **Output:** 21  **Explanation:** Ans = 7 + 7 + 7 = 21  One of the best strategies is:  1st week : stay at city 0, and play 7 days.  2nd week : fly from city 0 to city 1 on Monday, and play 7 days.  3rd week : fly from city 1 to city 2 on Monday, and play 7 days.  **Note:**   1. **N and K** are positive integers, which are in the range of [1, 100]. 2. In the matrix **days**, all the values are integers in the range of [0, 1]. 3. In the matrix **flights**, all the values are integers in the range [0, 7]. 4. You could stay at a city beyond the number of vacation days, but you should **work** on the extra days, which won't be counted as vacation days. 5. If you fly from the city A to the city B and take the vacation on that day, the deduction towards vacation days will count towards the vacation days of city B in that week. 6. We don't consider the impact of flight hours towards the calculation of vacation days.   C++1 (ETL)  class Solution {  public:  int vacation=0, city, week;  void dfs(vector<vector<int>>& flights, vector<vector<int>>& days, vector<bool> visited, int c, int w, int sum)  {  if(w==week)  {  vacation = max(vacation, sum);  return;  }  visited[c] = true;  for(int i=0;i<city; ++i)  {  if(i==c || (flights[c][i]==1 && !visited[i]))  dfs(flights, days, visited, i, w+1, sum+days[i][w]);  }  }  int maxVacationDays(vector<vector<int>>& flights, vector<vector<int>>& days) {  city = days.size(), week = days[0].size();  vector<bool> visited(city);  dfs(flights, days, visited, 0, 0, 0);  return vacation;  }  };  C++2 (dfs + dp)  class Solution {  public:  int dfs(vector<vector<int>>& flights, vector<vector<int>>& days, vector<vector<int>>& dp, int c, int w)  {  if(w==days[0].size())  {  return 0;  }  if(dp[c][w]!=INT\_MIN) return dp[c][w];  int maxDays = 0;  for(int i=0;i<flights.size(); ++i)  {  if(i==c || flights[c][i]==1)  {  int curDays = days[i][w] + dfs(flights, days, dp, i, w+1);  maxDays = max(maxDays, curDays);  }  }  dp[c][w] = maxDays;  return maxDays;  }  int maxVacationDays(vector<vector<int>>& flights, vector<vector<int>>& days) {  int city = days.size(), week = days[0].size();  vector<vector<int>> dp(city, vector<int>(week, INT\_MIN));  return dfs(flights, days, dp, 0, 0);  }  }; |
| 514. Freedom Trail  In the video game Fallout 4, the quest "Road to Freedom" requires players to reach a metal dial called the "Freedom Trail Ring", and use the dial to spell a specific keyword in order to open the door.  Given a string **ring**, which represents the code engraved on the outer ring and another string **key**, which represents the keyword needs to be spelled. You need to find the **minimum** number of steps in order to spell all the characters in the keyword.  Initially, the first character of the **ring** is aligned at 12:00 direction. You need to spell all the characters in the string **key** one by one by rotating the ring clockwise or anticlockwise to make each character of the string **key** aligned at 12:00 direction and then by pressing the center button.  At the stage of rotating the ring to spell the key character **key[i]**:   1. You can rotate the **ring** clockwise or anticlockwise **one place**, which counts as 1 step. The final purpose of the rotation is to align one of the string **ring's** characters at the 12:00 direction, where this character must equal to the character **key[i]**. 2. If the character **key[i]** has been aligned at the 12:00 direction, you need to press the center button to spell, which also counts as 1 step. After the pressing, you could begin to spell the next character in the key (next stage), otherwise, you've finished all the spelling.   **Example:**  https://leetcode.com/static/images/problemset/ring.jpg    **Input:** ring = "godding", key = "gd"  **Output:** 4  **Explanation:**  For the first key character 'g', since it is already in place, we just need 1 step to spell this character.   For the second key character 'd', we need to rotate the ring "godding" anticlockwise by two steps to make it become "ddinggo".  Also, we need 1 more step for spelling.  So the final output is 4.  **Note:**   1. Length of both ring and **key** will be in range 1 to 100. 2. There are only lowercase letters in both strings and might be some duplcate characters in both strings. 3. It's guaranteed that string **key** could always be spelled by rotating the string **ring**.   C++1 (TLE)  class Solution {  public:  int minRotateSteps(unordered\_map<char, vector<int>>& map, string& key, int ki, int curP, int rs)  {  if(ki==key.length()) return 0;  int gloMin = INT\_MAX;  for(int rP : map[key[ki]])  {  int dis = abs(rP-curP);  int curMin = min(dis, rs-dis);  curMin += minRotateSteps(map, key, ki+1, rP, rs);  gloMin = min(gloMin, curMin);  }  return gloMin;  }  int findRotateSteps(string ring, string key) {  unordered\_map<char, vector<int>> map;  for(int i=0; i<ring.length(); ++i)  map[ring[i]].push\_back(i);  return minRotateSteps(map, key, 0, 0, ring.length()) + key.length();  }  };  C++2  class Solution {  public:  int minRotateSteps(unordered\_map<char, vector<int>>& map, string& key, vector<vector<int>>& dp, int ki, int curP, int rs)  {  if(ki==key.length()) return 0;  if(dp[curP][ki] != INT\_MAX) return dp[curP][ki];  int gloMin = INT\_MAX;  for(int rP : map[key[ki]])  {  int dis = abs(rP-curP);  int curMin = min(dis, rs-dis);  curMin += minRotateSteps(map, key, dp, ki+1, rP, rs);  gloMin = min(gloMin, curMin);  }  dp[curP][ki] = gloMin;  return gloMin;  }  int findRotateSteps(string ring, string key) {  unordered\_map<char, vector<int>> map;  for(int i=0; i<ring.length(); ++i)  map[ring[i]].push\_back(i);  int k = key.length();  vector<vector<int>> dp(ring.length(), vector<int>(k, INT\_MAX));  return minRotateSteps(map, key, dp, 0, 0, ring.length()) + key.length();  }  };  Java1  public int findRotateSteps(String ring, String key) {  int n = ring.length();  int m = key.length();  int[][] dp = new int[m + 1][n];    for (int i = m - 1; i >= 0; i--) {  for (int j = 0; j < n; j++) {  dp[i][j] = Integer.MAX\_VALUE;  for (int k = 0; k < n; k++) {  if (ring.charAt(k) == key.charAt(i)) {  int diff = Math.abs(j - k);  int step = Math.min(diff, n - diff);  dp[i][j] = Math.min(dp[i][j], step + dp[i + 1][k]);  }  }  }  }    return dp[0][0] + m;  } |
|  |

### 广度优先(Breadth First Search)

### 启发式搜索(Heuristic Search)

## 以“相似或相同子问题”为核心的算法

### 递推

### 递归(Recursion)

|  |
| --- |
| 241. 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]  C++1 (9ms)  vector<int> diffWaysToCompute(string input) {  vector<int> res;  for(int i=0; i<input.length(); ++i)  {  char c = input[i];  if(ispunct(c))  for(auto a : diffWaysToCompute(input.substr(0, i)))  for(auto b : diffWaysToCompute(input.substr(i+1)))  res.push\_back(c=='+' ? a+b : (c=='-') ? a-b : a\*b);  }  return res.size()==0 ? vector<int>{stoi(input)} : res;  }  Java1  public List<Integer> diffWaysToCompute(String input) {  List<Integer> res = new LinkedList<Integer>();  for(int i=0; i<input.length(); ++i)  {  char ch = input.charAt(i);  if(!Character.isDigit(ch))  {  String left = input.substring(0, i);  String right = input.substring(i+1, input.length());  for(int a : diffWaysToCompute(left))  for(int b : diffWaysToCompute(right))  res.add((ch == '+') ? a+b :  (ch == '-') ? a-b :  a\*b);  }  }  if(res.size() == 0)  res.add(Integer.parseInt(input));  return res;  } |
| 96. 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 C++1 **Time Limit Exceeded** int numTrees(int n) {  if(n<=1) return 1;  int res = 0;  for(int i=1; i<=n; ++i)  {  int left = numTrees(i-1);  int right = numTrees(n-i);  res += left\*right;  }  return res;  }  C++2 (0ms) O(n2) O(n)space  int numTrees(int n) {  vector<int> dp(n+1);  dp[0] = dp[1] = 1;  for(int i=2; i<=n; ++i)  for(int j=0; j<i; ++j)  dp[i] += dp[j] \* dp[i-j-1];  return dp[n];  } |
| 486. 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 chooses 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 True representing player1 can win.  **Note:**   1. 1 <= length of the array <= 20. 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.   C++1 (3ms)  class Solution {  public:  bool canWin(int goal, vector<int>& nums, int i, int j)  {  if(goal <= 0) return true;  if(i <= j)  return canWin(goal-nums[i], nums, i+2, j) && canWin(goal-nums[i], nums, i+1, j-1)  || canWin(goal-nums[j], nums, i+1, j-1) && canWin(goal-nums[j], nums, i, j-2);  return false;  }  bool PredictTheWinner(vector<int>& nums) {  int goal = accumulate(nums.begin(), nums.end(), 0);  goal = (goal+1) / 2;  return canWin(goal, nums, 0, nums.size()-1);  }  };  C++2 DP |
| 494. 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.   C++1 (  class Solution {  public:  int subsetSum(vector<int>& nums, int target)  {  vector<int>dp(target+1);  dp[0]=1;  for(auto num : nums)  for(int i=target; i>=num; --i)  dp[i] += dp[i-num];  return dp[target];  }  int findTargetSumWays(vector<int>& nums, int S) {  int sum = accumulate(nums.begin(), nums.end(), 0);  return (sum<S || (sum+S)%2 != 0) ? 0 : subsetSum(nums, (sum+S)/2);  }  }; |
| 397. 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  C++1  int integerReplacement(int n) {  if(n==1) return 0;  if(n==INT\_MAX) return integerReplacement(n-1); //return 32;  if(n%2==0)  return integerReplacement(n/2) +1;  else  return min(integerReplacement(n+1), integerReplacement(n-1)) + 1;  } |
| Hr Russian Peasant Exponentiation We all know how to calculate ab using b operations by multiplying 1 by a a total of b times. The drawback to this method is that b can be large, which makes exponentiation very slow.  There is a well known method called Russian Peasant Multiplication that you can read about [here](https://www.hackerrank.com/external_redirect?to=http://lafstern.org/matt/col3.pdf). Now let's use this to raise some complex numbers to powers!  You're given q queries where each query consists of four integers: a, b, k, and m. For each query, calculate (a + b·i)k = c + d·i (where i is an imaginary unit) and then print the respective values of c mod m and d mod m as two space-separated integers on a new line.  **Input Format**  The first line contains a single integer, q, denoting the number of queries.  Each of the q subsequent lines describes a query in the form of four space-separated integers: a, b, k, and m (respectively).  **Constraints**   * 1 <= q <= 105 * 0 <= k <= 1018 * 2 <= m <= 109 * 0 <= a,b <= m   **Output Format**  For each query, print the two space-separated integers denoting the respective values of c mod m and d mod m on a new line.  **Sample Input**  3  2 0 9 1000  0 1 5 10  8 2 10 1000000000  **Sample Output**  512 0  0 1  880332800 927506432  **Explanation**  In the first query, we have a=2, b=0, k=9, m=1000. We calculate the following:   1. 29 = 512 2. i5 = i   C++1  vector<long> Multiply(vector<long>& ab, vector<long>& cd, long m){  vector<long> res(2);  res[0] = (ab[0]\*cd[0]%m - ab[1]\*cd[1]%m + m) % m;  res[1] = (ab[0]\*cd[1]%m + ab[1]\*cd[0]%m + m) % m;  return res;  }  vector<long> Cal(vector<long>& ab, long k, long m){  if(k==0){  vector<long>k0(2, 0);  k0[0] = 1;  return k0;  }  if(k==1) return ab;  vector<long> res = Cal(ab, k/2, m);  res = Multiply(res, res, m);  if(k%2 == 1)  res = Multiply(res, ab, m);  return res;  }  int main() {  int q;  vector<long> ab(2);  vector<long> res(2);  long k, m;  cin>>q;  for(int i=0; i<q; ++i){  cin>>ab[0];  cin>>ab[1];  cin>>k;  cin>>m;  res = Cal(ab, k, m);  cout<<to\_string(res[0])+" "+to\_string(res[1])<<endl;  }  return 0;  }  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int q = sc.nextInt();  long [] ab = new long[2];  long k;  long m;    for(int i=0; i<q; ++i){  ab[0] = sc.nextLong();  ab[1] = sc.nextLong();  k = sc.nextLong();  m = sc.nextLong();  long[] res = Cal(ab, k, m);  System.out.println(res[0]+" "+res[1]);  }  }    private static long[] Cal(long[] ab, long k, long m){  if(k==0) return new long[]{(long)1, (long)0};  if(k==1) return ab;  long [] res = Cal(ab, k/2, m);  res = Multiply(res, res, m);  if(k%2 == 1)  res = Multiply(res, ab, m);  return res;  }    private static long[] Multiply(long[] ab, long[] cd, long m){  long[] res = new long[2];  res[0] = (ab[0]\*cd[0] % m - ab[1]\*cd[1] % m + m) % m;  res[1] = (ab[0]\*cd[1] % m + ab[1]\*cd[0] % m) % m;  return res;  } |
|  |

#### Back Tracking

|  |
| --- |
| 77. Combinations  Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.  For example, If *n* = 4 and *k* = 2, a solution is:  [  [2,4],  [3,4],  [2,3],  [1,2],  [1,3],  [1,4],  ]  C++1 (76ms)  class Solution {  public:  void combination(vector<vector<int>>& res, vector<int>& row, int start, int n, int k)  {  if(k<=0)  res.push\_back(row);  else  {  for(int i=start; i<=n; ++i)  {  row.push\_back(i);  combination(res, row, i+1, n, k-1);  row.pop\_back();  }  }  }  vector<vector<int>> combine(int n, int k) {  vector<vector<int>> res;  vector<int> row;  combination(res, row, 1, n, k);  return res;  }  };  C++2 (89ms)  class Solution {  public:  void Helper(vector<vector<int>>& res, vector<int>& aRow, int n, int k, int start, int size)  {  if(k<=0)  {  res.push\_back(aRow);  return;  }  for(int i=start; i<=n; ++i)  {  aRow[size] = i;  Helper(res, aRow, n, k-1, i+1, size+1);  }  }  vector<vector<int>> combine(int n, int k) {  vector<vector<int>> res;  vector<int> aRow(k,0);  Helper(res, aRow, n, k, 1, 0);  return res;  }  };  C++3 (124ms)  vector<vector<int>> combine(int n, int k) {  vector<vector<int>> res;  vector<int> curr(k,0);  int i=0;  while(i>=0 && curr[0]<=(n-k)+1){  curr[i]++;  if(curr[i]>n) --i;  else if(i==k-1) res.push\_back(curr);  else{  i++;  curr[i]=curr[i-1];  }  }  return res;  }  Java1 (16ms)  public List<List<Integer>> combine(int n, int k) {  if(n<k || k==0)  {  return new LinkedList<>();  }  List<List<Integer>> res = combine(n-1, k-1);  if(res.size()==0)  res.add(new LinkedList<>(Arrays.asList(n)));  else  for(List<Integer> element : res)  element.add(n);  List<List<Integer>> withoutN = combine(n-1, k);  res.addAll(withoutN);  return res;  }  Java2  public List<List<Integer>> combine(int n, int k) {  if(n<k || k==0)  {  return new LinkedList<>();  }  List<List<Integer>> res = combine(n-1, k-1);  if(res.size()==0)  res.add(new LinkedList<>(Arrays.asList(n)));  else  res.forEach(e -> e.add(n));  List<List<Integer>> withoutN = combine(n-1, k);  res.addAll(withoutN);  return res;  } |
| 39. Combination Sum  Given a **set** of candidate numbers (***C***) **(without duplicates)** 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. * 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]  ]  C++1 (13ms)  class Solution {  public:  void checkCombinationSum(vector<vector<int>>& res, vector<int>& aRow, int start, vector<int>& candidates, int target)  {  if(start==candidates.size() || target<=0)  {  if(target==0) res.push\_back(aRow);  return;  }  for(int i=start; i<candidates.size(); ++i)  {  aRow.push\_back(candidates[i]);  checkCombinationSum(res, aRow, i, candidates, target-candidates[i]);  aRow.pop\_back();  }  }  vector<vector<int>> combinationSum(vector<int>& candidates, int target) {  vector<vector<int>> res;  vector<int> aRow;  checkCombinationSum(res, aRow, 0, candidates, target);  return res;  }  };  C++2 (13ms)  class Solution {  public:  void checkCombinationSum(vector<vector<int>>& res, vector<int>& aRow, int start, vector<int>& candidates, int target)  {  if(target==0)  {  res.push\_back(aRow);  return;  }  for(int i=start; i<candidates.size() && target-candidates[i]>=0; ++i)  {  aRow.push\_back(candidates[i]);  checkCombinationSum(res, aRow, i, candidates, target-candidates[i]);  aRow.pop\_back();  }  }  vector<vector<int>> combinationSum(vector<int>& candidates, int target) {  vector<vector<int>> res;  vector<int> aRow;  sort(candidates.begin(), candidates.end());  checkCombinationSum(res, aRow, 0, candidates, target);  return res;  }  };  Java1  public class Solution {  private List<List<Integer>> res;  public List<List<Integer>> combinationSum(int[] candidates, int target) {  res = new LinkedList<>();  List<Integer> arow = new ArrayList<>();  Arrays.sort(candidates);  collectCombSum(arow, candidates, target, 0);  return res;  }    private void collectCombSum(List<Integer> arow, int[] candidates, int target, int start){  if(target==0){  List<Integer> ares = new ArrayList<>(arow);  res.add(ares);  }  for(int i=start; i<candidates.length; ++i){  if(target-candidates[i] >= 0){  arow.add(candidates[i]);  collectCombSum(arow, candidates, target-candidates[i], i);  arow.remove(arow.size()-1);  }  else  break;  }  }  } |
| 40. 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. * 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]  ]  C++1  class Solution {  public:  void comSumBackTracking(vector<vector<int>>& res, vector<int>& aRow, vector<int>& candidates, int target, int start)  {  if(target==0)  {  res.push\_back(aRow);  return;  }  for(int i=start; i<candidates.size(); ++i)  {  if(i!=start && candidates[i]==candidates[i-1]) continue;  if(target-candidates[i] < 0) break;  aRow.push\_back(candidates[i]);  comSumBackTracking(res, aRow, candidates, target-candidates[i], i+1);  aRow.pop\_back();  }  }  vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {  vector<vector<int>> res;  vector<int> aRow;  sort(candidates.begin(), candidates.end());  comSumBackTracking(res, aRow, candidates, target, 0);  return res;  }  }; |
| 216. 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.  ***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]]  C++1 (6ms)  class Solution {  public:  void combination(vector<vector<int>>& res, vector<int>& aRow, int k, int n)  {  if(aRow.size() == k)  {  if(n==0) res.push\_back(aRow);  }  else if(aRow.size() < k)  {  for(int i= (aRow.empty()) ? 1 : aRow.back()+1; i<=9; ++i)  {  aRow.push\_back(i);  combination(res, aRow, k, n-i);  aRow.pop\_back();  }  }  }  vector<vector<int>> combinationSum3(int k, int n) {  vector<vector<int>> res;  vector<int> aRow;  combination(res, aRow, k, n);  return res;  }  };  C++2 (0ms)  class Solution {  public:  void Helper(vector<vector<int>>& res, vector<int>& acom, int k, int n)  {  if(acom.size()==k && n==0) res.push\_back(acom);  if(acom.size()<k)  {  for(int i=acom.empty()? 1: acom.back()+1; i<=9; ++i)  {  if(n-i<0) break;  acom.push\_back(i);  Helper(res, acom, k, n-i);  acom.pop\_back();  }  }  }  vector<vector<int>> combinationSum3(int k, int n) {  vector<vector<int>> res;  vector<int> acom;  Helper(res, acom, k, n);  return res;  }  };  Java1  public class Solution {    private List<List<Integer>> res;    public List<List<Integer>> combinationSum3(int k, int n) {  res = new ArrayList<>();  List<Integer> arow = new LinkedList<Integer>();  CShelper(arow, 1, k, n);  return res;  }    private void CShelper(List<Integer> arow, int start, int k, int n)  {  if(k==0 && n==0)  {  List<Integer> findOne = new LinkedList<Integer>(arow);  res.add(findOne);  return;  }  for(int i=start; i<=9; ++i)  {  arow.add(i);  CShelper(arow, i+1, k-1, n-i);  arow.remove(arow.size()-1);  }  }  } |
| 377. 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?  C++1 (3ms)  int combinationSum4(vector<int>& nums, int target) {  int comb[target+1] = {0};  comb[0] = 1;  sort(nums.begin(), nums.end());  for(int i=1; i<=target; ++i)  for(int j=0;j<nums.size() && nums[j]<=i; ++j)  comb[i] += comb[i-nums[j]];  return comb[target];  }  C++2 (12ms)  int combinationSum4(vector<int>& nums, int target) {  int dp[target+1] = {0};  dp[0] = 1;  for(int i=1; i<=target; ++i)  for(int num : nums)  if(i >= num)  dp[i] += dp[i - num];  return dp[target];  } |
| 526. 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 ith position (1 ≤ i ≤ N) in this array:   1. The number at the ith position is divisible by **i**. 2. **i** is divisible by the number at the ith 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.   C++1 (6ms)  class Solution {  public:  int CountArr(vector<int>& nums, int n)  {  if(n<=1) return 1;  int count=0;  for(int i=0; i<n; ++i)  {  if(nums[i]%n==0 || n%nums[i]==0)  {  swap(nums[i], nums[n-1]);  count += CountArr(nums, n-1);  swap(nums[i], nums[n-1]);  }  }  return count;  }  int countArrangement(int N) {  vector<int> nums(N);  for(int i=1; i<=N; ++i) nums[i-1] = i;  return CountArr(nums, N);  }  }; |
| 491. 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.   C++1 (296ms)  class Solution {  public:  void backTracking(vector<vector<int>>& res, vector<int>& sub, vector<int>& nums, int start, int end)  {  if(sub.size()>=2) res.push\_back(sub);  unordered\_map<int, int> map;  for(int i=start; i<end; ++i)  {  if(map[nums[i]] > 0) continue;  if(sub.size()==0 || nums[i]>=sub.back())  {  map[nums[i]]++;  sub.push\_back(nums[i]);  backTracking(res, sub, nums, i+1, end);  sub.pop\_back();  }  }  }  vector<vector<int>> findSubsequences(vector<int>& nums) {  vector<vector<int>> res;  vector<int> sub;  backTracking(res, sub, nums, 0, nums.size());  return res;  }  }; |
| 488. 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'.   C++1  class Solution {  public:  int countStep(string board, vector<int> hc)  {  board = removeConsecutiveBalls(board);  if(board=="#") return 0;  int res = 6;  for(int i=0, j=0; i<board.length(); ++i)  {  if(board[i] == board[j]) continue;  int need = 3-(i-j);  if(hc[board[j]] >= need)  {  hc[board[j]] -= need;  string aBoard = board.substr(0,j) + board.substr(i);  res = min(res, need + countStep(aBoard, hc));  hc[board[j]] += need;  }  j = i;  }  return res;  }  string removeConsecutiveBalls(string board)  {  for(int i=0, j=0; i<board.length(); ++i)  {  if(board[i] == board[j]) continue; //i might be out of length, reason for "#"  if(i-j>=3)  {  board = board.substr(0,j) + board.substr(i);  return removeConsecutiveBalls(board);  }  j = i;  }  return board;  }  int findMinStep(string board, string hand) {  vector<int> handBalls(128, 0);  for(char h : hand)  handBalls[h]++;  int ms = countStep(board+"#", handBalls);  return ms == 6 ? -1 : ms;  }  };  Testcase:  I tried case "RRWWRRBBRR", "WB". The test program gave the expected answer -1. However, I thought the answer might be 2. Because:  RRWWRRBBRR -> RRWWRRBBR[W]R -> RRWWRRBB[B]RWR -> RRWWRRRWR -> RRWWWR -> RRR -> empty |
| 473. 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 sticks 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.   C++1 (39ms)  class Solution {  public:  bool backTracking(vector<int>& nums, vector<int>& edges, int id, int target)  {  if(id==nums.size())  return edges[0]==edges[1] && edges[1]==edges[2] && edges[2]==edges[3];  for(int i=0; i<4; ++i)  {  if(edges[i]+nums[id]>target) continue;  edges[i] += nums[id];  if(backTracking(nums, edges, id+1, target)) return true;  edges[i] -= nums[id];  }  return false;  }  bool makesquare(vector<int>& nums) {  if(nums.size() < 4) return false;  int sum = accumulate(nums.begin(), nums.end(), 0);  if(sum%4 != 0) return false;  sort(nums.begin(), nums.end(), greater<int>());  vector<int> edges(4);  return backTracking(nums, edges, 0, sum/4);  }  };  C++2 (3ms)  class Solution {  public:  bool backTracking(vector<int>& nums, vector<int>& edges, int id, int target)  {  if(id==nums.size())  return edges[0]==edges[1] && edges[1]==edges[2] && edges[2]==edges[3];  for(int i=0; i<4; ++i)  {  if(edges[i]+nums[id]>target) continue;  int j=i;  while(--j>=0) if(edges[i] == edges[j]) break;  if(j != -1) continue;  edges[i] += nums[id];  if(backTracking(nums, edges, id+1, target)) return true;  edges[i] -= nums[id];  }  return false;  }  bool makesquare(vector<int>& nums) {  if(nums.size() < 4) return false;  int sum = accumulate(nums.begin(), nums.end(), 0);  if(sum%4 != 0) return false;  sort(nums.begin(), nums.end(), greater<int>());  vector<int> edges(4);  return backTracking(nums, edges, 0, sum/4);  }  }; |
| 17. 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.  http://upload.wikimedia.org/wikipedia/commons/thumb/7/73/Telephone-keypad2.svg/200px-Telephone-keypad2.svg.png  **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.  C++1  class Solution {  public:  void CollectLetters(vector<string>& num, string& digits, vector<string>& res, string arow, int start)  {  if(start >= digits.length() && arow.length()==digits.length())  res.push\_back(arow);  for(int i=start; i<digits.size(); ++i)  {  int d = digits[i]-'0';  for(int j=0; j<num[d].length(); ++j)  CollectLetters(num, digits, res, arow+num[d][j], i+1);  }  }  vector<string> letterCombinations(string digits) {  vector<string> num(10);  num[2] = "abc";  num[3] = "def";  num[4] = "ghi";  num[5] = "jkl";  num[6] = "mno";  num[7] = "pqrs";  num[8] = "tuv";  num[9] = "wxyz";  vector<string> res;  if(digits.empty()) return res;  string arow;  CollectLetters(num, digits, res, arow, 0);  return res;  }  };  C++2  vector<string> letterCombinations(string digits) {  vector<string> res;  if(digits.empty()) return res;  static const vector<string> phone = {"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};  res.push\_back("");  for(int i=0; i<digits.size(); ++i)  {  int n = digits[i]-'0';  if(n<2 || n>9) return vector<string>();  const string& candidate = phone[n];  if(candidate.empty()) continue;  vector<string> temp;  for(int j=0; j<candidate.size(); ++j)  {  for(int k=0; k<res.size(); ++k)  {  temp.push\_back(res[k] + candidate[j]);  }  }  res.swap(temp);  }  return res;  } |
| 113. 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]  ]  C++1 (26ms)  class Solution {  public:  void pathSumBacktracking(vector<vector<int>>& res, vector<int> aRow, TreeNode\* root, int sum)  {  if(root==NULL) return;  aRow.push\_back(root->val);  if(root->left==NULL && root->right==NULL)  {  if(sum-root->val==0)  res.push\_back(aRow);  }  else  {  pathSumBacktracking(res, aRow, root->left, sum-root->val);  pathSumBacktracking(res, aRow, root->right, sum-root->val);  }  }  vector<vector<int>> pathSum(TreeNode\* root, int sum) {  vector<vector<int>> res;  vector<int> aRow;  pathSumBacktracking(res, aRow, root, sum);  return res;  }  };  C++2 (9ms)  class Solution {  public:  void pathSumBacktracking(vector<vector<int>>& res, vector<int>& aRow, TreeNode\* root, int sum)  {  if(root==NULL) return;  aRow.push\_back(root->val);  sum -= root->val;  if(root->left==NULL && root->right==NULL && sum==0)  res.push\_back(aRow);  pathSumBacktracking(res, aRow, root->left, sum);  pathSumBacktracking(res, aRow, root->right, sum);  aRow.pop\_back();  }  vector<vector<int>> pathSum(TreeNode\* root, int sum) {  vector<vector<int>> res;  vector<int> aRow;  pathSumBacktracking(res, aRow, root, sum);  return res;  }  }; |
| 78. Subsets  Given a set of **distinct** integers, *nums*, return all possible subsets.  **Note:** 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],  []  ]  C++1 (9ms)  class Solution {  public:  void backTracking(vector<vector<int>>& res, vector<int>& aRow, vector<int>& nums, int start, int end)  {  if(start <= end)  {  res.push\_back(aRow);  for(int i=start; i<end; ++i)  {  aRow.push\_back(nums[i]);  backTracking(res, aRow, nums, i+1, end);  aRow.pop\_back();  }  }  }  vector<vector<int>> subsets(vector<int>& nums) {  vector<vector<int>> res;  vector<int> aRow;  backTracking(res, aRow, nums, 0, nums.size());  return res;  }  };  C++2  vector<vector<int>> subsets(vector<int>& nums) {  vector<vector<int>> res(1,vector<int>());  for(int i=0; i<nums.size(); ++i)  {  int n = res.size();  for(int j=0; j<n; ++j)  {  res.push\_back(res[j]);  res.back().push\_back(nums[i]);  }  }  return res;  }  Java1  public List<List<Integer>> subsets(int[] nums) {  List<List<Integer>> res = new ArrayList<>();  List<Integer> firstRow = new ArrayList<Integer>();  res.add(firstRow);  //Arrays.sort(nums);  for(int i=0; i<nums.length; ++i){  int len = res.size();  for(int j=0; j<len; ++j){  res.add(new ArrayList<Integer>(res.get(j)));  res.get(res.size()-1).add(nums[i]);  }  }  return res;  }  Java2  public class Solution {    private List<List<Integer>> res;    public List<List<Integer>> subsets(int[] nums) {  res = new LinkedList<>();  List<Integer> arow = new LinkedList<Integer>();  collectSubsets(arow, nums, 0);  return res;  }    private void collectSubsets(List<Integer> arow, int[] nums, int start){  List<Integer> ares = new LinkedList<Integer>(arow);  res.add(ares);  for(int i=start; i<nums.length; ++i){  arow.add(nums[i]);  collectSubsets(arow, nums, i+1);  arow.remove(arow.size()-1);  }  }  } |
| 90. Subsets II  Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets.  **Note:** 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],  []  ]  C++1 (back tracking)  class Solution {  public:  void CollectSubsets(vector<vector<int>>& res, vector<int> aRow, int start, vector<int> nums)  {  if(start >= nums.size()) return;  for(int i=start; i<nums.size(); ++i)  {  if(i!=start && nums[i]==nums[i-1]) continue;  aRow.push\_back(nums[i]);  res.push\_back(aRow);  CollectSubsets(res, aRow, i+1, nums);  aRow.pop\_back();  }  }  vector<vector<int>> subsetsWithDup(vector<int>& nums) {  vector<vector<int>> res;  vector<int> aRow;  res.push\_back(aRow);  sort(nums.begin(), nums.end());  CollectSubsets(res, aRow, 0, nums);  return res;  }  };  C++2 (iteration)  vector<vector<int>> subsetsWithDup(vector<int>& nums) {  sort(nums.begin(), nums.end());  vector<vector<int>> res(1, vector<int>());  vector<int> aRow;  int size = 1;  for(int i=0; i<nums.size(); ++i)  {  int start = (i>0)&&(nums[i]==nums[i-1]) ? size : 0;  size = res.size();  for(int j=start; j<size; ++j)  {  res.push\_back(res[j]);  res.back().push\_back(nums[i]);  }  }  return res;  } |
| 79. 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.  C++1  class Solution {  public:  bool hasWord(vector<vector<char>>& board, int i, int j, string& word, int wi)  {  if(wi == word.size()) return true;  if(i<0 || i>=board.size() || j<0 || j>=board[0].size()) return false;  if(board[i][j]!=word[wi]) return false;  board[i][j] = 0;  bool res = hasWord(board,i-1,j,word,wi+1)  || hasWord(board,i+1,j,word,wi+1)  || hasWord(board,i,j-1,word,wi+1)  || hasWord(board,i,j+1,word,wi+1);  board[i][j] = word[wi];  return res;  }  bool exist(vector<vector<char>>& board, string word) {  int h=board.size();  int w = h>0 ? board[0].size() : 0;  if(word.empty()) return false;  for(int i=0; i<h; ++i)  {  for(int j=0; j<w; ++j)  if(hasWord(board, i, j, word, 0)) return true;  }  return false;  }  }; |
| 52. N-Queens II  Follow up for N-Queens problem.  Now, instead outputting board configurations, return the total number of distinct solutions.  https://leetcode.com/static/images/problemset/8-queens.png  class Solution {  public:  int DFS(vector<vector<bool>>& mem, int row, int n)  {  if(row==n) return 1;  int res = 0;  for(int i=0; i<n; ++i)  {  if(mem[0][i]==false && mem[1][row+i]==false && mem[2][n+i-row]==false)  {  mem[0][i]=mem[1][row+i]=mem[2][n+i-row]=true;  res += DFS(mem, row+1, n);  mem[0][i]=mem[1][row+i]=mem[2][n+i-row]=false;  }  }  return res;  }  int totalNQueens(int n) {  vector<vector<bool>> mem(3, vector<bool>(2\*n-1));  // vector<int> col(n), main(2\*n-1), anti(2\*n-1); // int is more effective than bool  return DFS(mem, 0, n);  }  }; |
| Hr Building a List Chan has decided to make a list of all possible combinations of letters of a given string S. If there are two strings with the same set of characters, print the lexicographically smallest arrangement of the two strings.  abc acb cab bac bca  all the above strings' lexicographically smallest string is abc.  Each character in the string S is unique. Your task is to print the entire list of Chan's in lexicographic order.  for string *abc*, the list in lexicographic order is given below  a ab abc ac b bc c  **Input Format**  The first line contains the number of test cases T. T testcases follow.  Each testcase has 2 lines. The first line is an integer N ( the length of the string).  The second line contains the string S.  **Output Format**  For each testcase, print the entire list of combinations of string S, with each combination of letters in a newline.  **Constraints**  0< T< 50  1< N< 16  string S contains only small alphabets(a-z)  **Sample Input**  2  2  ab  3  xyz  **Sample Output**  a  ab  b  x  xy  xyz  xz  y  yz  z  **Explanation**  In the first case we have ab, the possibilities are a, ab and b. Similarly, all combination of characters of xyz.  Java1  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  int T = sc.nextInt();  for(int i=0; i<T; ++i){  int N = sc.nextInt();  String s = sc.next();  char[] c = s.toCharArray();  Arrays.sort(c);  BackTracking(0, "", c);  }  }    private static void BackTracking(int start, String arow, char[] c){  for(int i=start; i<c.length; ++i){  System.out.println(arow+c[i]);  BackTracking(i+1, arow+c[i], c);  }  } |
|  |

### 贪心法(Greedy)

|  |
| --- |
| 406. 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]]  C++1 (52ms) O(nlog(n))  class Solution {  public:  static bool myComp(pair<int,int> a, pair<int,int> b)  {  return a.first>b.first || (a.first==b.first && a.second<b.second);  }  vector<pair<int, int>> reconstructQueue(vector<pair<int, int>>& people) {  sort(people.begin(), people.end(), myComp);  vector<pair<int, int>>res;  for(auto person : people)  res.insert(res.begin()+person.second, person);  return res;  }  };  C++2 (76ms) O(nlog(n))  vector<pair<int, int>> reconstructQueue(vector<pair<int, int>>& people) {  sort(people.begin(), people.end(), [](pair<int,int> a, pair<int,int> b)  {  return a.first>b.first || (a.first==b.first && a.second<b.second);  });  vector<pair<int,int>> res;  for(auto person : people)  {  res.insert(res.begin() + person.second, person);  }  return res;  } |
| 452. 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 104 balloons.  An arrow can be shot up exactly vertically from different points along the x-axis. A balloon with xstart and xend bursts by an arrow shot at x if xstart ≤ x ≤ xend. 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).  C++1  public:  static bool myCompare(pair<int, int> a, pair<int, int> b)  {  return a.first < b.first;  }  int findMinArrowShots(vector<pair<int, int>>& points) {  int res = 1, start=INT\_MIN, end=INT\_MAX;  sort(points.begin(), points.end(), myCompare);  for(auto point : points)  {  if(point.first > end)  {  res ++;  start = point.first;  end = point.second;  }  else  {  start = point.first;  end = min(end, point.second);  }  }  return points.empty() ? 0 : res;  }  };  C++2  int findMinArrowShots(vector<pair<int, int>>& points) {  if(points.empty()) return 0;  sort(points.begin(), points.end(), [](pair<int, int>& a, pair<int, int>& b){  return a.first < b.first || (a.first==b.first && a.second < b.second);  });  int res = 1, minB = points[0].first, maxB =points[0].second;  for(int i=1; i<points.size(); ++i)  {  if(points[i].first > maxB)  res++;  maxB = (points[i].first > maxB)? points[i].second : min(maxB, points[i].second);  minB = points[i].first;  }  return res;  } |
| 312. 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  C++1  int maxCoins(vector<int>& nums) {  int n = nums.size();  nums.insert(nums.begin(), 1);  nums.push\_back(1);  int dp[n+2][n+2] = {0};  for(int len=1; len<=n; ++len)  for(int left=1; left<=n-len+1; ++left)  for(int mid=left, right=left+len-1; mid<=right; ++mid)  dp[left][right] = max(dp[left][right], nums[left-1]\*nums[mid]\*nums[right+1] + dp[left][mid-1] + dp[mid+1][right]);  return dp[1][n];  } |
| 436. 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.  C++1 (99ms)  vector<int> findRightInterval(vector<Interval>& intervals) {  map<int, int> map;  for(int i=0; i<intervals.size(); ++i)  map[intervals[i].start] = i;  vector<int> res(intervals.size());  for(int i=0; i<intervals.size(); ++i)  {  auto it = map.lower\_bound(intervals[i].end);  res[i] = (it != map.end()) ? it->second : -1;  }  return res;  } |
| 435. 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-overlapping.  **Example 3:**  **Input:** [ [1,2], [2,3] ]  **Output:** 0  **Explanation:** You don't need to remove any of the intervals since they're already non-overlapping.  C++1 (9ms) O(nlog(n))  class Solution {  public:  static bool compEnd(Interval a, Interval b)  {  return a.end < b.end;  }  int eraseOverlapIntervals(vector<Interval>& intervals) {  int n = intervals.size();  if(n==0) return 0;  sort(intervals.begin(), intervals.end(), compEnd);  int nonOverlapNum = 1, End = intervals[0].end;  for(auto interval : intervals)  {  if(interval.start >= End)  {  End = interval.end;  nonOverlapNum++;  }  }  return n - nonOverlapNum;  }  }; |
| 495. 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 poisoned 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 status 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 poisoned.  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.   C++1 (63ms)  int findPoisonedDuration(vector<int>& timeSeries, int duration) {  int res = 0, n = timeSeries.size();  if(n==0) return 0;  for(int i=1; i<n; ++i)  {  int temp = timeSeries[i]-timeSeries[i-1];  res += temp<duration ? temp : duration;  }  return res + duration;  } |
| Codility [MaxNonoverlappingSegments](https://codility.com/demo/results/trainingBEB6D5-HC8/)  Located on a line are N segments, numbered from 0 to N − 1, whose positions are given in zero-indexed arrays A and B. For each I (0 ≤ I < N) the position of segment I is from A[I] to B[I] (inclusive). The segments are sorted by their ends, which means that B[K] ≤ B[K + 1] for K such that 0 ≤ K < N − 1.  Two segments I and J, such that I ≠ J, are *overlapping* if they share at least one common point. In other words, A[I] ≤ A[J] ≤ B[I] or A[J] ≤ A[I] ≤ B[J].  We say that the set of segments is *non-overlapping* if it contains no two overlapping segments. The goal is to find the size of a non-overlapping set containing the maximal number of segments.  For example, consider arrays A, B such that:  A[0] = 1 B[0] = 5 A[1] = 3 B[1] = 6 A[2] = 7 B[2] = 8 A[3] = 9 B[3] = 9 A[4] = 9 B[4] = 10  The segments are shown in the figure below.  https://codility-frontend-prod.s3.amazonaws.com/media/task_static/max_nonoverlapping_segments/static/images/auto/68b279360bc48af61d9d3bdfbe1d30fe.png  The size of a non-overlapping set containing a maximal number of segments is 3. For example, possible sets are {0, 2, 3}, {0, 2, 4}, {1, 2, 3} or {1, 2, 4}. There is no non-overlapping set with four segments.  Write a function:  int solution(vector<int> &A, vector<int> &B);  that, given two zero-indexed arrays A and B consisting of N integers, returns the size of a non-overlapping set containing a maximal number of segments.  For example, given arrays A, B shown above, the function should return 3, as explained above.  Assume that:   * N is an integer within the range [0..30,000]; * each element of arrays A, B is an integer within the range [0..1,000,000,000]; * A[I] ≤ B[I], for each I (0 ≤ I < N); * B[K] ≤ B[K + 1], for each K (0 ≤ K < N − 1).   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **struct** Segment{  **int** left;  **int** right;  Segment(**int** x, **int** y) : left(x), right(y) {}  };  **int** **solution**(**vector**<**int**> &A, **vector**<**int**> &B) {  // write your code in C++14 (g++ 6.2.0)  **int** n = B.size();  **if**(n==0) **return** 0;  Segment **seg**(A[0], B[0]);  **int** res = 1;  **for**(**int** i=1; i<n; ++i)  {  **if**(A[i]<=seg.right)  {  seg.left = max(seg.left, A[i]);  seg.right = min(seg.right, B[i]);  }  **else**  {  res++;  seg.left = A[i];  seg.right = B[i];  }  }  **return** res;  }  Testcase  extreme\_empty\_and\_single  empty and single element  [▶](https://codility.com/demo/results/trainingBEB6D5-HC8/)  small\_functional  many overlapping  [▶](https://codility.com/demo/results/trainingBEB6D5-HC8/)  small\_non\_overlapping  all non-overlapping  [▶](https://codility.com/demo/results/trainingBEB6D5-HC8/)  small\_all\_overlapping  small functional  [▶](https://codility.com/demo/results/trainingBEB6D5-HC8/)  small\_random\_same\_length  small random, length = ~40  [▶](https://codility.com/demo/results/trainingBEB6D5-HC8/)  medium\_random\_differ\_length  medium random, length = ~300  [▶](https://codility.com/demo/results/trainingBEB6D5-HC8/)  large\_points  all points, length = ~30,000  [▶](https://codility.com/demo/results/trainingBEB6D5-HC8/)  large\_random\_many\_overlapping  large random, length = ~30,000  [▶](https://codility.com/demo/results/trainingBEB6D5-HC8/)  large\_random\_few\_overlapping  large random, length = ~30,000  [▶](https://codility.com/demo/results/trainingBEB6D5-HC8/)  extreme\_large  large size of intervals, length = ~30,000 |
| Codility [TieRopes](https://codility.com/demo/results/trainingYKGP7Q-NCM/)  There are N ropes numbered from 0 to N − 1, whose lengths are given in a zero-indexed array A, lying on the floor in a line. For each I (0 ≤ I < N), the length of rope I on the line is A[I].  We say that two ropes I and I + 1 are *adjacent*. Two adjacent ropes can be tied together with a knot, and the length of the tied rope is the sum of lengths of both ropes. The resulting new rope can then be tied again.  For a given integer K, the goal is to tie the ropes in such a way that the number of ropes whose length is greater than or equal to K is maximal.  For example, consider K = 4 and array A such that:  A[0] = 1 A[1] = 2 A[2] = 3 A[3] = 4 A[4] = 1 A[5] = 1 A[6] = 3  The ropes are shown in the figure below.  https://codility-frontend-prod.s3.amazonaws.com/media/task_static/tie_ropes/static/images/auto/f13a51b17fba1ea9b8ea7fd37006f767.png  We can tie:   * rope 1 with rope 2 to produce a rope of length A[1] + A[2] = 5; * rope 4 with rope 5 with rope 6 to produce a rope of length A[4] + A[5] + A[6] = 5.   After that, there will be three ropes whose lengths are greater than or equal to K = 4. It is not possible to produce four such ropes.  Write a function:  int solution(int K, vector<int> &A);  that, given an integer K and a non-empty zero-indexed array A of N integers, returns the maximum number of ropes of length greater than or equal to K that can be created.  For example, given K = 4 and array A such that:  A[0] = 1 A[1] = 2 A[2] = 3 A[3] = 4 A[4] = 1 A[5] = 1 A[6] = 3  the function should return 3, as explained above.  Assume that:   * N is an integer within the range [1..100,000]; * K is an integer within the range [1..1,000,000,000]; * each element of array A is an integer within the range [1..1,000,000,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**int** K, **vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** res = 0, count=0;  **for**(**int** a : A)  {  count += a;  **if**(count>=K)  {  res++;  count = 0;  }  }  **return** res;  }  Testcase  single  single element  [▶](https://codility.com/demo/results/trainingYKGP7Q-NCM/)  double  two elements  [▶](https://codility.com/demo/results/trainingYKGP7Q-NCM/)  small\_functional  small functional tests  [▶](https://codility.com/demo/results/trainingYKGP7Q-NCM/)  small\_random  small random sequences length = ~100  [▶](https://codility.com/demo/results/trainingYKGP7Q-NCM/)  medium\_random  chaotic medium sequences length = ~5,000  [▶](https://codility.com/demo/results/trainingYKGP7Q-NCM/)  large\_range  large range test, length = ~100,000  [▶](https://codility.com/demo/results/trainingYKGP7Q-NCM/)  large\_answer  test with large answer, length = ~100,000  [▶](https://codility.com/demo/results/trainingYKGP7Q-NCM/)  small\_answer  test with large answer, length = ~100,000 |
| 56. 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].  C++1  vector<Interval> merge(vector<Interval>& intervals) {  vector<Interval> res;  if(intervals.empty()) return res;  sort(intervals.begin(), intervals.end(), [](Interval a, Interval b)  {  return a.start < b.start;  });  res.push\_back(intervals[0]);  for(int i=1; i<intervals.size(); ++i)  {  if(intervals[i].start <= res.back().end)  {  res.back().end = max(res.back().end, intervals[i].end);  }  else  res.push\_back(intervals[i]);  }  return res;  } |
| 646. Maximum Length of Pair Chain  You are given n pairs of numbers. In every pair, the first number is always smaller than the second number.  Now, we define a pair (c, d) can follow another pair (a, b) if and only if b < c. Chain of pairs can be formed in this fashion.  Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs. You can select pairs in any order.  **Example 1:**  **Input:** [[1,2], [2,3], [3,4]]  **Output:** 2  **Explanation:** The longest chain is [1,2] -> [3,4]  **Note:**   1. The number of given pairs will be in the range [1, 1000].   C++1 int findLongestChain(vector<vector<int>>& pairs) {  sort(pairs.begin(), pairs.end(), [](vector<int> a, vector<int> b) {  return a[1]<b[1];  });    int res = 0, right = INT\_MIN;  for(vector<int> pair : pairs)  {  if(pair[0] > right)  {  res++;  right = pair[1];  }  }  return res;  }  C++2  static bool myfunction (vector<int> a, vector<int> b) { return a[1]<b[1]; }    int findLongestChain(vector<vector<int>>& pairs) {  sort(pairs.begin(), pairs.end(), myfunction);    int res = 0, right = INT\_MIN;  for(vector<int> pair : pairs)  {  if(pair[0] > right)  {  res++;  right = pair[1];  }  }  return res;  }  C++3  struct myclass {  bool operator() (vector<int> a, vector<int> b) { return a[1]<b[1]; }  } myobject;    int findLongestChain(vector<vector<int>>& pairs) {  sort(pairs.begin(), pairs.end(), myobject);    int res = 0, right = INT\_MIN;  for(vector<int> pair : pairs)  {  if(pair[0] > right)  {  res++;  right = pair[1];  }  }  return res;  }  Java1  public int findLongestChain(int[][] pairs) {  Arrays.sort(pairs, (int[] a, int[] b) -> a[1] - b[1] );    int res = 0, right = Integer.MIN\_VALUE;  for(int i=0; i<pairs.length; ++i){  if(pairs[i][0] > right){  res++;  right = pairs[i][1];  }  }  return res;  }  Java2  public class Solution {  public int findLongestChain(int[][] pairs) {  Arrays.sort(pairs, new SortbyRight());    int res = 0, right = Integer.MIN\_VALUE;  for(int i=0; i<pairs.length; ++i){  if(pairs[i][0] > right){  res++;  right = pairs[i][1];  }  }  return res;  }  }  class SortbyRight implements Comparator<int[]>  {  // Used for sorting in ascending order of right edge  public int compare(int[] a, int[] b)  {  return a[1] - b[1];  }  } |
|  |

## 动态规划(Dynamic Programming)

1005. Fast Food

|  |
| --- |
| 419. Battleships in a Board  Given an 2D board, count how many battleships are in it. The battleships are represented with 'X's, empty slots are represented with '.'s. You may assume the following rules:   * You receive a valid board, made of only battleships or empty slots. * Battleships can only be placed horizontally or vertically. In other words, they can only be made of the shape 1xN (1 row, N columns) or Nx1 (N rows, 1 column), where N can be of any size. * At least one horizontal or vertical cell separates between two battleships - there are no adjacent battleships.   **Example:**  X..X  ...X  ...X  In the above board there are 2 battleships.  **Invalid Example:**  ...X  XXXX  ...X  This is an invalid board that you will not receive - as battleships will always have a cell separating between them.  **Follow up:** Could you do it in **one-pass**, using only **O(1) extra memory** and **without modifying** the value of the board?  C++1 (6ms) O(mn)  int countBattleships(vector<vector<char>>& board) {  int row=board.size();  int col = (row==0) ? 0 : board[0].size();  int res=0;  for(int i=0; i<row; ++i)  {  for(int j=0; j<col; ++j)  {  if(board[i][j]=='.') continue;  if(i>0 && board[i-1][j]=='X') continue;  if(j>0 && board[i][j-1]=='X') continue;  res++;  }  }  return res;  }  Java1 (3ms)  public int countBattleships(char[][] board) {  int row = board.length;  int col = (row>0) ? board[0].length : 0;  int res =0;  for(int i=0; i<row; ++i)  {  for(int j=0; j<col; ++j)  {  if(board[i][j] == '.') continue;  if(i>0 && board[i-1][j] == 'X') continue;  if(j>0 && board[i][j-1] == 'X') continue;  res ++;  }  }  return res;  } |
| 338. 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.   **Hint:**   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?   C++1 (85ms)  vector<int> countBits(int num) {  vector<int> res(1,0);  while(num>0)  {  int size= res.size();  for(int i=0; i<size && num-->0; ++i)  {  res.push\_back(res[i]+1);  }  }  return res;  }  C++2 (84ms)  vector<int> countBits(int num) {  vector<int> res(num+1,0);  for(int i=1; i<=num; ++i)  {  res[i] = res[i&(i-1)] + 1;  }  return res;  }  C++3 (88ms)  vector<int> countBits(int num) {  vector<int> res(num+1, 0);  if(num==0) return res;  res[1] = 1;  if(num==1) return res;  int div = 1;  for(int i=2; i<=num; ++i)  {  res[i] = res[i%div] + 1;  if(i%div == 0)  div \*= 2;  }  return res;  }  Java1 (2ms)  public int[] countBits(int num) {  int[] res = new int[num+1];  for(int i=1; i<=num; ++i)  {  res[i] = res[i&(i-1)] + 1;  }  return res;  }  Java2 (3ms)  public int[] countBits(int num) {  int[] res = new int[num+1];  if(num==0) return res;  res[1] = 1;  if(num==0) return res;  int div = 1;  for(int i=2; i<=num; ++i)  {  res[i] = res[i%div] + 1;  if(i%div == 0)  div \*= 2;  }  return res;  } |
| 357. Count Numbers with Unique Digits  Given a **non-negative** integer n, count all numbers with unique digits, x, where 0 ≤ x < 10n.  **Example:** Given n = 2, return 91. (The answer should be the total numbers in the range of 0 ≤ x < 100, excluding [11,22,33,44,55,66,77,88,99])  **Hint:**   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 10n. 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].   C++1 (0ms)  int countNumbersWithUniqueDigits(int n) {  n = (n>10) ? 10 : n;  int res = 1, dig = 9, cur = 9;  for(int i=1; i<=n; ++i)  {  res += cur;  cur \*= dig--;  }  return res;  }  C++2  int countNumbersWithUniqueDigits(int n) {  int dp[11] = {1};  dp[1] = 10;  n = min(10, n);  for(int i=2; i<=n; ++i)  {  dp[i] = 9;  int temp = 9;  for(int j=1; j<i; ++j)  dp[i] \*= temp--;  dp[i] += dp[i-1];  }  return dp[n];  }  Java1  public int countNumbersWithUniqueDigits(int n) {  int[] dp = new int[11];  dp[0] = 1;  dp[1] = 10;  n = Math.min(n, 10);  for(int i=2; i<=n; ++i)  {  dp[i] = 9;  int temp = 9;  for(int j=1; j<i; ++j)  dp[i] \*= temp--;  dp[i] += dp[i-1];  }  return dp[n];  } |
| 377. 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?  C++1 (3ms)  int combinationSum4(vector<int>& nums, int target) {  int comb[target+1] = {0};  comb[0] = 1;  sort(nums.begin(), nums.end());  for(int i=1; i<=target; ++i)  for(int j=0;j<nums.size() && nums[j]<=i; ++j)  comb[i] += comb[i-nums[j]];  return comb[target];  }  C++2 (12ms)  int combinationSum4(vector<int>& nums, int target) {  int dp[target+1] = {0};  dp[0] = 1;  for(int i=1; i<=target; ++i)  for(int num : nums)  if(i >= num)  dp[i] += dp[i - num];  return dp[target];  } |
| 96. 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 C++1 **Time Limit Exceeded** int numTrees(int n) {  if(n<=1) return 1;  int res = 0;  for(int i=1; i<=n; ++i)  {  int left = numTrees(i-1);  int right = numTrees(n-i);  res += left\*right;  }  return res;  }  C++2 (0ms) O(n2) O(n)space  int numTrees(int n) {  vector<int> dp(n+1);  dp[0] = dp[1] = 1;  for(int i=2; i<=n; ++i)  for(int j=0; j<i; ++j)  dp[i] += dp[j] \* dp[i-j-1];  return dp[n];  } |
| 62. Unique Paths  A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).  The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).  How many possible unique paths are there?  http://leetcode.com/wp-content/uploads/2014/12/robot_maze.png  Above is a 3 x 7 grid. How many possible unique paths are there?  **Note:** *m* and *n* will be at most 100.  C++1 (0ms) O(mn)  int uniquePaths(int m, int n) {  if(n<1) return 0;  vector<int> dp(n, 1);  for(int i=1; i<m; ++i)  for(int j=1; j<n; ++j)  dp[j] += dp[j-1];  return dp[n-1];  }  C++2  int uniquePaths(int m, int n) {  if(m>n) return uniquePaths(n, m);  double res=1;  for(int i=m-1; i>0; --i)  {  res \*= (double)(n-1+i)/i;  }  return round(res);  }  Java1  public int uniquePaths(int m, int n) {  if(m < n)  return uniquePaths(n, m);  if( n<= 0) return 0;  int[] dp = new int[n];  Arrays.fill(dp, 1);  for(int i=1; i<m; ++i)  for(int j=1; j<n; ++j)  dp[j] += dp[j-1];  return dp[n-1];  } |
| 516. 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".  C++1 (49ms)  int longestPalindromeSubseq(string s) {  int n = s.length();  if(n==0) return 0;  vector<vector<int>> dp(n, vector<int>(n));  for(int i=n-1; i>=0; --i)  {  dp[i][i] = 1;  for(int j=i+1; j<n; ++j)  dp[i][j] = (s[i]==s[j]) ? dp[i+1][j-1] + 2  : max(dp[i+1][j], dp[i][j-1]);  }  return dp[0][n-1];  }  C++2 (TLE)  class Solution {  public:  int RescursivePalindrome(string s, vector<vector<int>>& dp, int start, int end)  {  if(dp[start][end] != 0) return dp[start][end];  if(start > end) return 0;  if(start == end) return 1;  dp[start][end] = (s[start] == s[end]) ? RescursivePalindrome(s, dp, start+1, end-1) + 2  : max(RescursivePalindrome(s, dp, start+1, end), RescursivePalindrome(s, dp, start, end-1));  return dp[start][end];  }  int longestPalindromeSubseq(string s) {  int n = s.length();  if(n==0) return 0;  vector<vector<int>> dp(n, vector<int>(n));  return RescursivePalindrome(s, dp, 0, n-1);  }  };  Testcases:   |  |  |  | | --- | --- | --- | | Id | Testcase | Result | | 1 | [] | 0 | | 2 | [a] | 1 | | 3 | "fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffgggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg" | 494 | |
| 416. 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.  C++1 (49ms)  bool canPartition(vector<int>& nums) {  int sum = accumulate(nums.begin(), nums.end(), 0);  if(sum%2 != 0) return false;  sum /= 2;  vector<bool> dp(sum+1);  dp[0] = true;  for(auto num : nums)  for(int i = sum; i >= num; i--)  dp[i] = dp[i] || dp[i - num];  return dp[sum];  }  C++2( 9ms)  bool canPartition(vector<int>& nums) {  bitset<5001> bits(1);  int sum = accumulate(nums.begin(), nums.end(), 0);  for(auto num : nums) bits |= bits << num;  return !(sum & 1) && bits[sum>>1];  } |
| 494. 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.   C++1 (386ms)  class Solution {  public:  int TargetSum(vector<int>& nums, int S, int i)  {  if(i == nums.size())  return S==0 ? 1 : 0;  int res = 0;  res += TargetSum(nums, S+nums[i], i+1);  res += TargetSum(nums, S-nums[i], i+1);  return res;  }  int findTargetSumWays(vector<int>& nums, int S) {  return TargetSum(nums, S, 0);  }  };  C++2 (3ms) DP  class Solution {  public:  int subsetSum(vector<int>& nums, int target)  {  vector<int>dp(target+1);  dp[0]=1;  for(auto num : nums)  for(int i=target; i>=num; --i)  dp[i] += dp[i-num];  return dp[target];  }  int findTargetSumWays(vector<int>& nums, int S) {  int sum = accumulate(nums.begin(), nums.end(), 0);  return (sum<S || (sum+S)%2 != 0) ? 0 : subsetSum(nums, (sum+S)/2);  }  }; |
| Codility [NumberSolitaire](https://codility.com/demo/results/trainingHMETHB-MKC/)  A game for one player is played on a board consisting of N consecutive squares, numbered from 0 to N − 1. There is a number written on each square. A non-empty zero-indexed array A of N integers contains the numbers written on the squares. Moreover, some squares can be marked during the game.  At the beginning of the game, there is a pebble on square number 0 and this is the only square on the board which is marked. The goal of the game is to move the pebble to square number N − 1.  During each turn we throw a six-sided die, with numbers from 1 to 6 on its faces, and consider the number K, which shows on the upper face after the die comes to rest. Then we move the pebble standing on square number I to square number I + K, providing that square number I + K exists. If square number I + K does not exist, we throw the die again until we obtain a valid move. Finally, we mark square number I + K.  After the game finishes (when the pebble is standing on square number N − 1), we calculate the result. The result of the game is the sum of the numbers written on all marked squares.  For example, given the following array:  A[0] = 1 A[1] = -2 A[2] = 0 A[3] = 9 A[4] = -1 A[5] = -2  one possible game could be as follows:   * the pebble is on square number 0, which is marked; * we throw 3; the pebble moves from square number 0 to square number 3; we mark square number 3; * we throw 5; the pebble does not move, since there is no square number 8 on the board; * we throw 2; the pebble moves to square number 5; we mark this square and the game ends.   The marked squares are 0, 3 and 5, so the result of the game is 1 + 9 + (−2) = 8. This is the maximal possible result that can be achieved on this board.  Write a function:  int solution(vector<int> &A);  that, given a non-empty zero-indexed array A of N integers, returns the maximal result that can be achieved on the board represented by array A.  For example, given the array  A[0] = 1 A[1] = -2 A[2] = 0 A[3] = 9 A[4] = -1 A[5] = -2  the function should return 8, as explained above.  Assume that:   * N is an integer within the range [2..100,000]; * each element of array A is an integer within the range [−10,000..10,000].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size();  **vector**<**int**> dp(n, -1e9);  dp[0]=A[0];  **for**(**int** i=1; i<n; ++i)  {  **for**(**int** j=i-1; j>=0 && j>=i-6; --j)  dp[i] = max(dp[i], A[i]+dp[j]);  }  **return** dp[n-1];  }  C++2  int solution(vector<int> &A) {  int N(A.size());  vector<int> die6(6, A[0]);  for (int i = 1; i < N; ++i)  die6[i % 6] = \*max\_element(die6.begin(), die6.end()) + A[i];  return die6[(N-1) % 6];  }  Testcase  extreme  two or three fields  [▶](https://codility.com/demo/results/trainingHMETHB-MKC/)  simple  simple test  [▶](https://codility.com/demo/results/trainingHMETHB-MKC/)  medium\_all\_negative  all values negative, length = ~1,000  [▶](https://codility.com/demo/results/trainingHMETHB-MKC/)  medium\_monotonic  monotonic sequence, length = ~1,000  [▶](https://codility.com/demo/results/trainingHMETHB-MKC/)  medium\_random  random sequence of values, length = ~1,000  [▶](https://codility.com/demo/results/trainingHMETHB-MKC/)  big\_all\_negative  all values negative, length = ~100,000  [▶](https://codility.com/demo/results/trainingHMETHB-MKC/)  big\_random  random sequence of values, length = ~100,000  [▶](https://codility.com/demo/results/trainingHMETHB-MKC/)  extreme\_answers  maximal and minimal answers |
| 64. 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.  C++1  int minPathSum(vector<vector<int>>& grid) {  int h = grid.size();  int w = (h!=0) ? grid[0].size() : 0;  if(w==0) return 0;  vector<int> dp(w, INT\_MAX);  dp[0] = 0;  for(int i=0; i<h; ++i)  {  dp[0] += grid[i][0];  for(int j=1; j<w; ++j)  dp[j] = min(dp[j-1], dp[j]) + grid[i][j];  }  return w==0 ? 0 : dp[w-1];  }  C++2  int minPathSum(vector<vector<int>>& grid) {  if(grid.empty()) return 0;  int m = grid.size();  int n = grid[0].size();  vector<vector<int>> res(m+1, vector<int>(n+1, INT\_MAX));  res[0][1] = 0;  for(int i=1; i<=m; ++i)  {  for(int j=1; j<=n; ++j)  {  res[i][j] = min(res[i-1][j], res[i][j-1]) + grid[i-1][j-1];  }  }  return res[m][n];  }  Java1  public int minPathSum(int[][] grid) {  int m=grid.length, n=grid[0].length;  int[] dp = new int[n];  for(int i=1; i<n; ++i)  dp[i] = Integer.MAX\_VALUE;  for(int i=0; i<m; ++i)  {  dp[0] += grid[i][0];  for(int j=1; j<n; ++j)  dp[j] = Math.min(dp[j], dp[j-1]) + grid[i][j];  }  return dp[n-1];  } |
| 474. 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".  C++1  class Solution {  public:  struct Str01{  int zero;  int one;  Str01(int a, int b) : zero(a), one(b) {}  };  Str01 count01(string str)  {  int a=0, b=0;  for(char c : str)  {  if(c=='0') a++;  else b++;  }  return Str01(a, b);  }  int findMaxForm(vector<string>& strs, int m, int n) {  vector<vector<int>> dp(m+1, vector<int>(n+1));  for(string str : strs)  {  Str01 str01 = count01(str);  for(int i=m; i>=str01.zero; --i)  {  for(int j=n; j>=str01.one; --j)  dp[i][j] = max(dp[i][j], 1+dp[i-str01.zero][j-str01.one]);  }  }  return dp[m][n];  }  }; |
| 552. Student Attendance Record II  Given a positive integer **n**, return the number of all possible attendance records with length n, which will be regarded as rewardable. The answer may be very large, return it after mod 109 + 7.  A student attendance record is a string that only contains the following three characters:   1. **'A'** : Absent. 2. **'L'** : Late. 3. **'P'** : Present.   A record is regarded as rewardable if it doesn't contain **more than one 'A' (absent)** or **more than two continuous 'L' (late)**.  **Example 1:**  **Input:** n = 2  **Output:** 8  **Explanation:**  There are 8 records with length 2 will be regarded as rewardable:  "PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"  Only "AA" won't be regarded as rewardable owing to more than one absent times.  **Note:** The value of **n** won't exceed 100,000.  C++1  int checkRecord(int n) {  int mod = 1000000007;  int dp[n+1][2][3];  for(int i=0; i<2; ++i)  for(int j=0; j<3; ++j)  dp[0][i][j] = 1;  for(int p=1; p<=n; ++p)  {  for(int a=0; a<2; ++a)  {  for(int L=0; L<3; ++L)  {  int val = dp[p-1][a][2];  if(a>0) val = (val + dp[p-1][a-1][2]) % mod;  if(L>0) val = (val + dp[p-1][a][L-1]) % mod;  dp[p][a][L] = val;  }  }  }  return dp[n][1][2];  } |
| 279. 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.  C++1 (162ms)  int numSquares(int n) {  vector<int> dp(n+1);  for(int len=1; len<=n; ++len)  {  int minLen = INT\_MAX;  for(int i=1; i\*i<=len; ++i)  minLen = min(minLen, dp[len-i\*i] + 1);  dp[len] = minLen;  }  return dp[n];  }  C++2 (172ms)  int numSquares(int n) {  vector<int> dp(n+1, INT\_MAX);  dp[0] = 0;  for(int len=1; len<=n; ++len)  for(int i=1; i\*i<=len; ++i)  dp[len] = min(dp[len], dp[len-i\*i] + 1);  return dp[n];  }  C++3 (306ms)  int numSquares(int n) {  if(n<=0) return 0;  vector<int> perfS(1,0);  int len, minSN;  while(perfS.size()<=n)  {  len = perfS.size();  minSN = INT\_MAX;  for(int i=1; i\*i<=len; ++i)  {  minSN = min(minSN, perfS[len-i\*i]+1);  }  perfS.push\_back(minSN);  }  return perfS[n];  }  C++4 (12ms)  int numSquares(int n) {  if(n<=0) return 0;  static vector<int> perfS(1,0);  int len, minSN;  while(perfS.size()<=n)  {  len = perfS.size();  minSN = INT\_MAX;  for(int i=1; i\*i<=len; ++i)  {  minSN = min(minSN, perfS[len-i\*i]+1);  }  perfS.push\_back(minSN);  }  return perfS[n];  } |
| 388. 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. subdir2contains 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.  [Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question.  C++1  int lengthLongestPath(string input) {  vector<int>dp(1,0);  int res = 0;  stringstream ss(input);  string block;  while(getline(ss, block, '\n'))  {  int i = block.find\_first\_not\_of('\t')+1;  if(i>=dp.size())  dp.push\_back(0);  dp[i] = dp[i-1]+block.size()-i+2; //include "/"  if(block.find('.') != string::npos)  res = max(res, dp[i]-1);  }  return res;  }  C++2  int lengthLongestPath(string input) {  vector<int> dp;  int longest = 0;  stringstream ss(input);  string cur;  while(getline(ss, cur, '\n')){  int i= cur.find\_first\_not\_of('\t');  if(i+1 > dp.size())  dp.push\_back(0);  dp[i] = (i-1>=0 ? dp[i-1] : 0) + cur.size()-i+1;  if(cur.find('.') != string::npos)  longest = max(longest, dp[i]-1);  }  return longest;  } |
| 375. 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 at least you need to have to guarantee a **win**.  C++1  class Solution {  public:  int MinToWin(vector<vector<int>>& dp, int start, int end)  {  if(start >= end) return 0;  if(dp[start][end]!=0) return dp[start][end];  int minV = INT\_MAX;  for(int i=start; i<end; ++i)  {  int maxLoc = i + max(MinToWin(dp, start, i-1), MinToWin(dp, i+1,end));  minV = min(minV, maxLoc);  }  dp[start][end]= minV;  return dp[start][end];  }  int getMoneyAmount(int n) {  vector<vector<int>> dp(n+1, vector<int>(n+1));  return MinToWin(dp, 1, n);  }  }; |
| 36. Valid Sudoku  Determine if a Sudoku is valid, according to: [Sudoku Puzzles - The Rules](http://sudoku.com.au/TheRules.aspx).  The Sudoku board could be partially filled, where empty cells are filled with the character '.'.  http://upload.wikimedia.org/wikipedia/commons/thumb/f/ff/Sudoku-by-L2G-20050714.svg/250px-Sudoku-by-L2G-20050714.svg.png  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.  C++1  bool isValidSudoku(vector<vector<char>>& board) {  vector<vector<bool>> row(9, vector<bool>(9));  vector<vector<bool>> col(9, vector<bool>(9));  vector<vector<bool>> square(9, vector<bool>(9));  for(int i=0; i<9; ++i)  {  for(int j=0; j<9; ++j)  {  if(board[i][j] != '.')  {  int num = board[i][j]-'1';  int squth = i/3\*3 + j/3;  if(row[i][num] || col[j][num] || square[squth][num])  return false;  row[i][num] = col[j][num] = square[squth][num] = true;  }  }  }  return true;  } |
| 417. 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 parentheses in above matrix).  C++1  class Solution {  public:  void dfsOceans(vector<vector<int>>& matrix, vector<vector<bool>>& visited, int i, int j, int height)  {  if(i<0 || i>=matrix.size() || j<0 || j>=matrix[0].size() || visited[i][j]) return;  if(matrix[i][j] < height) return;  visited[i][j] = true;  dfsOceans(matrix, visited, i-1, j, matrix[i][j]);  dfsOceans(matrix, visited, i+1, j, matrix[i][j]);  dfsOceans(matrix, visited, i, j-1, matrix[i][j]);  dfsOceans(matrix, visited, i, j+1, matrix[i][j]);  }  vector<pair<int, int>> pacificAtlantic(vector<vector<int>>& matrix) {  vector<pair<int, int>> res;  int h = matrix.size();  int w = (h>0) ? matrix[0].size() : 0;  vector<vector<bool>> Pacific(h, vector<bool>(w));  vector<vector<bool>> Atlantic(h, vector<bool>(w));  for(int i=0; i<h; ++i)  {  dfsOceans(matrix, Pacific, i, 0, matrix[i][0]);  dfsOceans(matrix, Atlantic, i, w-1, matrix[i][w-1]);  }  for(int j=0; j<w; ++j)  {  dfsOceans(matrix, Pacific, 0, j, matrix[0][j]);  dfsOceans(matrix, Atlantic, h-1, j, matrix[h-1][j]);  }  for(int i=0; i<h; ++i)  for(int j=0; j<w; ++j)  if(Pacific[i][j] && Atlantic[i][j])  res.push\_back(make\_pair(i, j));  return res;  }  }; |
| 568. Maximum Vacation Days  LeetCode wants to give one of its best employees the option to travel among **N** cities to collect algorithm problems. But all work and no play makes Jack a dull boy, you could take vacations in some particular cities and weeks. Your job is to schedule the traveling to maximize the number of vacation days you could take, but there are certain rules and restrictions you need to follow.  **Rules and restrictions:**   1. You can only travel among **N** cities, represented by indexes from 0 to N-1. Initially, you are in the city indexed 0 on **Monday**. 2. The cities are connected by flights. The flights are represented as a **N\*N** matrix (not necessary symmetrical), called **flights** representing the airline status from the city i to the city j. If there is no flight from the city i to the city j, **flights[i][j] = 0**; Otherwise, **flights[i][j] = 1**. Also, **flights[i][i] = 0** for all i. 3. You totally have **K** weeks (**each week has 7 days**) to travel. You can only take flights at most once **per day** and can only take flights on each week's **Monday** morning. Since flight time is so short, we don't consider the impact of flight time. 4. For each city, you can only have restricted vacation days in different weeks, given an **N\*K** matrix called **days** representing this relationship. For the value of **days[i][j]**, it represents the maximum days you could take vacation in the city **i** in the week **j**.   You're given the **flights** matrix and **days** matrix, and you need to output the maximum vacation days you could take during **K** weeks.  **Example 1:**  **Input:**flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[1,3,1],[6,0,3],[3,3,3]]  **Output:** 12  **Explanation:**  Ans = 6 + 3 + 3 = 12.  One of the best strategies is:  1st week : fly from city 0 to city 1 on Monday, and play 6 days and work 1 day.  (Although you start at city 0, we could also fly to and start at other cities since it is Monday.)  2nd week : fly from city 1 to city 2 on Monday, and play 3 days and work 4 days.  3rd week : stay at city 2, and play 3 days and work 4 days.  **Example 2:**  **Input:**flights = [[0,0,0],[0,0,0],[0,0,0]], days = [[1,1,1],[7,7,7],[7,7,7]]  **Output:** 3  **Explanation:**  Ans = 1 + 1 + 1 = 3.  Since there is no flights enable you to move to another city, you have to stay at city 0 for the whole 3 weeks.  For each week, you only have one day to play and six days to work.  So the maximum number of vacation days is 3.  **Example 3:**  **Input:**flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[7,0,0],[0,7,0],[0,0,7]]  **Output:** 21  **Explanation:** Ans = 7 + 7 + 7 = 21  One of the best strategies is:  1st week : stay at city 0, and play 7 days.  2nd week : fly from city 0 to city 1 on Monday, and play 7 days.  3rd week : fly from city 1 to city 2 on Monday, and play 7 days.  **Note:**   1. **N and K** are positive integers, which are in the range of [1, 100]. 2. In the matrix **days**, all the values are integers in the range of [0, 1]. 3. In the matrix **flights**, all the values are integers in the range [0, 7]. 4. You could stay at a city beyond the number of vacation days, but you should **work** on the extra days, which won't be counted as vacation days. 5. If you fly from the city A to the city B and take the vacation on that day, the deduction towards vacation days will count towards the vacation days of city B in that week. 6. We don't consider the impact of flight hours towards the calculation of vacation days.   C++1 (ETL)  class Solution {  public:  int vacation=0, city, week;  void dfs(vector<vector<int>>& flights, vector<vector<int>>& days, vector<bool> visited, int c, int w, int sum)  {  if(w==week)  {  vacation = max(vacation, sum);  return;  }  visited[c] = true;  for(int i=0;i<city; ++i)  {  if(i==c || (flights[c][i]==1 && !visited[i]))  dfs(flights, days, visited, i, w+1, sum+days[i][w]);  }  }  int maxVacationDays(vector<vector<int>>& flights, vector<vector<int>>& days) {  city = days.size(), week = days[0].size();  vector<bool> visited(city);  dfs(flights, days, visited, 0, 0, 0);  return vacation;  }  };  C++2 (dfs + dp)  class Solution {  public:  int dfs(vector<vector<int>>& flights, vector<vector<int>>& days, vector<vector<int>>& dp, int c, int w)  {  if(w==days[0].size())  {  return 0;  }  if(dp[c][w]!=INT\_MIN) return dp[c][w];  int maxDays = 0;  for(int i=0;i<flights.size(); ++i)  {  if(i==c || flights[c][i]==1)  {  int curDays = days[i][w] + dfs(flights, days, dp, i, w+1);  maxDays = max(maxDays, curDays);  }  }  dp[c][w] = maxDays;  return maxDays;  }  int maxVacationDays(vector<vector<int>>& flights, vector<vector<int>>& days) {  int city = days.size(), week = days[0].size();  vector<vector<int>> dp(city, vector<int>(week, INT\_MIN));  return dfs(flights, days, dp, 0, 0);  }  }; |
| 467. 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 "zab" in the string s.  C++1  int findSubstringInWraproundString(string p) {  if(p.empty()) return 0;  vector<int> dp(26);  dp[p[0]-'a'] = 1;  for(int i=1, curMax=1, gap = 'a'-'z'; i<p.length(); ++i)  {  curMax = (i!=0 && (p[i]-p[i-1]==1 || p[i]-p[i-1]==gap)) ? curMax+1 : 1;  dp[p[i]-'a'] = max(dp[p[i]-'a'], curMax);  }  return accumulate(dp.begin(), dp.end(), 0);  } |
| 139. Word Break  Given a **non-empty** string *s* and a dictionary *wordDict* containing a list of **non-empty** words, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words. You may assume the dictionary does not contain duplicate words.  For example, given *s* = "leetcode", *dict* = ["leet", "code"].  Return true because "leetcode" can be segmented as "leet code".  C++1  bool wordBreak(string s, vector<string>& wordDict) {  int n=s.length();  vector<bool> dp(n+1);  dp[0] = true;  for(int i=1; i<=n; ++i)  for(int j=i-1; j>=0; --j)  if(dp[j]==true)  {  string sub = s.substr(j, i-j);  if(find(wordDict.begin(), wordDict.end(), sub) != wordDict.end())  {  dp[i] = true;  break;  }  }  return dp[n];  }  C++2  bool wordBreak(string s, vector<string>& wordDict) {  int n=s.length();  vector<bool> dp(n+1);  dp[0] = true;  unordered\_map<string, bool>map;  for(string word : wordDict)  map[word] = true;  for(int i=1; i<=n; ++i)  for(int j=i-1; j>=0; --j)  if(dp[j]==true)  {  string sub = s.substr(j, i-j);  if(map[sub] == true)  {  dp[i] = true;  break;  }  }  return dp[n];  } |
| 221. Maximal Square  Given a 2D binary matrix filled with 0's and 1's, find the largest square containing only 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.  C++1  int maximalSquare(vector<vector<char>>& matrix) {  int h = matrix.size();  int w = h>0 ? matrix[0].size() : 0;  vector<int> dp(w+1);  int edge=0, topLeft=0;  for(int i=0; i<h; ++i)  {  for(int j=1; j<=w; ++j)  {  int temp = dp[j];  if(matrix[i][j-1]=='1')  {  dp[j] = min(dp[j], min(dp[j-1], topLeft)) + 1;  edge = max(edge, dp[j]);  }  else  dp[j] = 0;  topLeft = temp;  }  }  return edge\*edge;  } |
| 576. Out of Boundary Paths  There is an **m** by **n** grid with a ball. Given the start coordinate **(i,j)** of the ball, you can move the ball to **adjacent** cell or cross the grid boundary in four directions (up, down, left, right). However, you can **at most** move **N** times. Find out the number of paths to move the ball out of grid boundary. The answer may be very large, return it after mod 109 + 7.  **Example 1:**  **Input:**m = 2, n = 2, N = 2, i = 0, j = 0  **Output:** 6  **Explanation:**  https://leetcode.com/static/images/problemset/out_of_boundary_paths_1.png  **Example 2:**  **Input:**m = 1, n = 3, N = 3, i = 0, j = 1  **Output:** 12  **Explanation:**  https://leetcode.com/static/images/problemset/out_of_boundary_paths_2.png  **Note:**   1. Once you move the ball out of boundary, you cannot move it back. 2. The length and height of the grid is in range [1,50]. 3. N is in range [0,50].   C++1 (TLE)  int findPaths(int m, int n, int N, int i, int j) {  if(i<0 || i==m || j<0 || j==n)  {  if(N>=0) return 1;  return 0;  }  if(N<=0) return 0;  int res = 0;  if(i>=0) res += findPaths(m,n,N-1,i-1,j);  if(i<m) res = (res + findPaths(m,n,N-1,i+1,j)) % bound;  if(j>=0) res = (res + findPaths(m,n,N-1,i,j-1)) % bound;  if(j<n) res = (res + findPaths(m,n,N-1,i,j+1)) % bound;  dp[i][j][N] = res;  return res;  }  C++2 (TLE)  class Solution {  public:  int DFS(int m, int n, int N, int i, int j, vector<vector<vector<int>>>& dp)  {  if(i<0 || i==m || j<0 || j==n)  {  if(N>=0) return 1;  return 0;  }  if(N<=0 || i>N && m-i>N && j>N && n-j>N)  {  dp[i][j][N] = -1;  return 0;  }  if(dp[i][j][N]==-1) return 0;  if(dp[i][j][N] > 0) return dp[i][j][N];    int res = 0;  if(i>=0) res += findPaths(m,n,N-1,i-1,j);  if(i<m) res = (res + findPaths(m,n,N-1,i+1,j)) % bound;  if(j>=0) res = (res + findPaths(m,n,N-1,i,j-1)) % bound;  if(j<n) res = (res + findPaths(m,n,N-1,i,j+1)) % bound;  dp[i][j][N] = res;  return res;  }  const int bound = (1e9) + 7;  int findPaths(int m, int n, int N, int i, int j) {  vector<vector<vector<int>>>dp(m, vector<vector<int>>(n, vector<int>(N+1)));  return DFS(m,n,N,i,j,dp);  }  };  C++3  int findPaths(int m, int n, int N, int i, int j) {  const int bound = (1e9) + 7;  vector<vector<vector<int>>>dp(m, vector<vector<int>>(n, vector<int>(N+1)));  for(int Ni=1; Ni<=N; ++Ni)  for(int mi=0; mi<m; ++mi)  for(int nj=0; nj<n; ++nj)  {  int res = (mi==0)? 1 : dp[mi-1][nj][Ni-1];  res = (res + ((mi==m-1)? 1 : dp[mi+1][nj][Ni-1])) % bound;  res = (res + ((nj==0)? 1 : dp[mi][nj-1][Ni-1])) % bound;  res = (res + ((nj==n-1)? 1 : dp[mi][nj+1][Ni-1])) % bound;  dp[mi][nj][Ni] = res;  }  return dp[i][j][N];  } |
| 43. Multiply Strings  Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2.  **Note:**   1. The length of both num1 and num2 is < 110. 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.   C++1  class Solution {  public:  string multiplyOneDigit(string num2, int n)  {  string res = "";  int carry=0;  for(int i=num2.length()-1; i>=0; --i)  {  int a =num2[i]-'0';  int mult = a\*n +carry;  carry = mult/10;  res += to\_string(mult%10);  }  return carry>0 ? res + to\_string(carry) : res;  }  string add(string pre, string cur, int len)  {  string res = pre.substr(0, len);  int i=len, j=0, carry=0;  int pLen= pre.length(), cLen = cur.length();  while(i<pLen || j<cLen || carry>0)  {  int a = i<pLen ? pre[i++]-'0' : 0;  int b = j<cLen ? cur[j++]-'0' : 0;  int sum = a+b+carry;  carry = sum/10;  res += to\_string(sum%10);  }  return res;  }  string multiply(string num1, string num2) {  string res="";  if(num1.empty()) return num2;  if(num2.empty()) return num1;  if(num1[0]=='0' || num2[0]=='0') return "0";  int n = num1.length();  for(int i=n-1; i>=0; --i)  {  string str = multiplyOneDigit(num2, num1[i]-'0');  res = add(res, str, n-i-1);  }  reverse(res.begin(), res.end());  return res;  }  };  C++2  string multiply(string num1, string num2) {  long ns1=num1.size(), ns2=num2.size();  if(ns1==0 || ns2==0) return "0";  int n1, n2, sum, carry;  vector<int> dp(ns1+ns2, 0);  for(long i=0; i<ns1; ++i)  {  n1 = num1[ns1-i-1] - '0';  carry=0;  for(long j=0; j<ns2; ++j)  {  n2 = num2[ns2-j-1] - '0';  sum = n1\*n2 + dp[i+j] + carry;  carry = sum/10;  dp[i+j] = sum%10;  }  if(carry>0) dp[i+ns2] += carry;  }  long start=ns1+ns2-1;  while(start>=0 && dp[start]==0) start--;  if(start==-1) return "0";  string res = "";  for(long i=start; i>=0; --i) res += '0' + dp[i];  return res;  } |
| 322. 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.  C++1 (TLE)  int coinChange(vector<int>& coins, int amount) {  if(amount==0) return 0;  int res = INT\_MAX;  for(int coin : coins)  {  if(amount-coin >= 0)  {  int x = coinChange(coins, amount-coin);  if(x!=-1)  res = min(res, x) + 1;  }  }  return res==INT\_MAX ? -1 : res;  }  C++2 (TLE)  int coinChange(vector<int>& coins, int amount) {  if(amount == 0) return 0;  vector<int>dp(amount+1, INT\_MAX);  for(int coin : coins)  if(coin<=amount)  dp[coin] = 1;  for(int i=1; i<=amount; ++i)  {  if(dp[i] == 1) continue;  for(int j=i-1; j>=1; --j)  if(dp[j]!=INT\_MAX && dp[i-j]!=INT\_MAX)  dp[i] = min(dp[i], dp[j]+dp[i-j]);  }  return dp[amount]==INT\_MAX ? -1 : dp[amount];  }  C++3  int coinChange(vector<int>& coins, int amount) {  vector<int>dp(amount+1, amount+1);  dp[0] = 0;  for(int coin : coins)  for(int i=coin; i<=amount; ++i)  dp[i] = min(dp[i], dp[i-coin]+1);  return dp[amount]>amount ? -1 : dp[amount];  } |
| 583. Delete Operation for Two Strings  Given two words *word1* and *word2*, find the minimum number of steps required to make *word1* and *word2* the same, where in each step you can delete one character in either string.  **Example 1:**  **Input:** "sea", "eat"  **Output:** 2  **Explanation:** You need one step to make "sea" to "ea" and another step to make "eat" to "ea".  **Note:**   1. The length of given words won't exceed 500. 2. Characters in given words can only be lower-case letters.   C++1 (TLE)  class Solution {  public:  int LCS(string word1, int i1, string word2, int i2)  {  if(i1==0 || i2==0) return 0;  else if(word1[i1-1]==word2[i2-1])  return 1 + LCS(word1, i1-1, word2, i2-1);  else  return max(LCS(word1, i1-1, word2, i2), LCS(word1, i1, word2, i2-1));  }  int minDistance(string word1, string word2) {  return word1.length()+word2.length() - 2\*LCS(word1, word1.length(), word2, word2.length());  }  };  C++2 (with memory)  class Solution {  public:  int LCS(string word1, int i1, string word2, int i2, vector<vector<int>>& mem)  {  if(i1==0 || i2==0) return 0;  if(mem[i1][i2]>0) return mem[i1][i2];  if(word1[i1-1]==word2[i2-1])  mem[i1][i2] = 1 + LCS(word1, i1-1, word2, i2-1, mem);  else  mem[i1][i2] = max(LCS(word1, i1-1, word2, i2, mem), LCS(word1, i1, word2, i2-1, mem));  return mem[i1][i2];  }  int minDistance(string word1, string word2) {  int n1=word1.length(), n2=word2.length();  vector<vector<int>> mem(n1+1, vector<int>(n2+1));  return n1 + n2 - 2\*LCS(word1, n1, word2, n2, mem);  }  };  C++3  int minDistance(string word1, string word2) {  int n1=word1.length(), n2=word2.length();  vector<vector<int>> dp(n1+1, vector<int>(n2+1));  for(int i1=1; i1<=n1; ++i1)  for(int i2=1; i2<=n2; ++i2)  if(word1[i1-1]==word2[i2-1])  dp[i1][i2] = 1+dp[i1-1][i2-1];  else  dp[i1][i2] = max(dp[i1-1][i2], dp[i1][i2-1]);  return n1 + n2 - 2\*dp[n1][n2];  }  C++4  int minDistance(string word1, string word2) {  int n1=word1.length(), n2=word2.length();  vector<vector<int>> dp(n1+1, vector<int>(n2+1));  for(int i1=0; i1<=n1; ++i1)  for(int i2=0; i2<=n2; ++i2)  if(i1==0 || i2==0)  dp[i1][i2] = i1+i2;  else if(word1[i1-1]==word2[i2-1])  dp[i1][i2] = dp[i1-1][i2-1];  else  dp[i1][i2] = 1 + min(dp[i1-1][i2], dp[i1][i2-1]);  return dp[n1][n2];  }  C++5  int minDistance(string word1, string word2) {  int n1=word1.length(), n2=word2.length();  vector<int> dp(n2+1);  for(int i1=0; i1<=n1; ++i1)  {  vector<int> temp = dp;  for(int i2=0; i2<=n2; ++i2)  if(i1==0 || i2==0)  dp[i2] = i1+i2;  else if(word1[i1-1]==word2[i2-1])  dp[i2] = temp[i2-1];  else  dp[i2] = 1 + min(dp[i2], dp[i2-1]);  }  return dp[n2];  } |
| 546. Remove Boxes  Given several boxes with different colors represented by different positive numbers.  You may experience several rounds to remove boxes until there is no box left. Each time you can choose some continuous boxes with the same color (composed of k boxes, k >= 1), remove them and get k\*k points. Find the maximum points you can get.  **Example 1:** Input:  [1, 3, 2, 2, 2, 3, 4, 3, 1]  Output:  23  Explanation:  [1, 3, 2, 2, 2, 3, 4, 3, 1]  ----> [1, 3, 3, 4, 3, 1] (3\*3=9 points)  ----> [1, 3, 3, 3, 1] (1\*1=1 points)  ----> [1, 1] (3\*3=9 points)  ----> [] (2\*2=4 points)  **Note:** The number of boxes n would not exceed 100.  C++1 (TLE)  int removeBoxes(vector<int>& boxes) {  int res = 0;  for(int i=0; i<boxes.size()-1; ++i)  {  if(boxes[i]<=0) continue;  int j=i+1, count=1, temp=boxes[i];  boxes[i] = -i;  while(j<boxes.size() && (boxes[j]<=0 || boxes[j]==temp))  {  if(boxes[j]==temp)  {  count++;  boxes[j] = -i;  }  j++;  }  res = max(res, count\*count + removeBoxes(boxes));  for(int b=i; b<j; ++b)  if(boxes[b]==-i)  boxes[b] = temp;  }  return res;  }  C++2  class Solution {  public:  int DFS(vector<int>& boxes, int dp[100][100][100], int start, int end, int len)  {  if(start>end) return 0;  if(dp[start][end][len] > 0) return dp[start][end][len];    while(start<end && boxes[end-1]==boxes[end])  {  end--;  len++;  }  dp[start][end][len] = DFS(boxes, dp, start, end-1, 0) + (len+1)\*(len+1); //at least = 1;  for(int i=start; i<end; ++i)  {  if(boxes[i]==boxes[end])  dp[start][end][len] = max(dp[start][end][len], DFS(boxes, dp, start, i, len+1)+DFS(boxes, dp, i+1,end-1, 0)); //i+1~end-1 has used  }  return dp[start][end][len];  }  int removeBoxes(vector<int>& boxes) {  int n=boxes.size();  int dp[100][100][100] = {0};  return DFS(boxes, dp, 0, n-1, 0);  }  }; |
| 91. 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.  C++1  int numDecodings(string s) {  int n = s.length();  if(n==0) return 0;  vector<int> dp(n+1);  dp[n] = 1;  dp[n-1] = s[n-1]=='0' ? 0 : 1;  for(int i=n-2; i>=0; --i)  if(s[i] != '0')  dp[i] = stoi(s.substr(i,2))<=26 ? dp[i+1]+dp[i+2] : dp[i+1];  return dp[0];  } |
| 312. 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  C++1  int maxCoins(vector<int>& nums) {  int n = nums.size();  nums.insert(nums.begin(), 1);  nums.push\_back(1);  int dp[n+2][n+2] = {0};  for(int len=1; len<=n; ++len)  for(int left=1; left<=n-len+1; ++left)  for(int mid=left, right=left+len-1; mid<=right; ++mid)  dp[left][right] = max(dp[left][right], nums[left-1]\*nums[mid]\*nums[right+1] + dp[left][mid-1] + dp[mid+1][right]);  return dp[1][n];  } |
| 638. Shopping Offers  In LeetCode Store, there are some kinds of items to sell. Each item has a price.  However, there are some special offers, and a special offer consists of one or more different kinds of items with a sale price.  You are given the each item's price, a set of special offers, and the number we need to buy for each item. The job is to output the lowest price you have to pay for **exactly** certain items as given, where you could make optimal use of the special offers.  Each special offer is represented in the form of an array, the last number represents the price you need to pay for this special offer, other numbers represents how many specific items you could get if you buy this offer.  You could use any of special offers as many times as you want.  **Example 1:**  **Input:** [2,5], [[3,0,5],[1,2,10]], [3,2]  **Output:** 14  **Explanation:**  There are two kinds of items, A and B. Their prices are $2 and $5 respectively.  In special offer 1, you can pay $5 for 3A and 0B  In special offer 2, you can pay $10 for 1A and 2B.  You need to buy 3A and 2B, so you may pay $10 for 1A and 2B (special offer #2), and $4 for 2A.  **Example 2:**  **Input:** [2,3,4], [[1,1,0,4],[2,2,1,9]], [1,2,1]  **Output:** 11  **Explanation:**  The price of A is $2, and $3 for B, $4 for C.  You may pay $4 for 1A and 1B, and $9 for 2A ,2B and 1C.  You need to buy 1A ,2B and 1C, so you may pay $4 for 1A and 1B (special offer #1), and $3 for 1B, $4 for 1C.  You cannot add more items, though only $9 for 2A ,2B and 1C.  **Note:**   1. There are at most 6 kinds of items, 100 special offers. 2. For each item, you need to buy at most 6 of them. 3. You are **not** allowed to buy more items than you want, even if that would lower the overall price.   C++1  class Solution {  public:  int sumPrice(vector<int>& price, vector<int>& needs)  {  int res = 0;  for(int i=0; i<price.size(); ++i)  res += price[i]\*needs[i];  return res;  }    int shoppingOffers(vector<int>& price, vector<vector<int>>& special, vector<int>& needs) {  int res = sumPrice(price, needs);  for(auto offer : special)  {  vector<int> clone(needs.begin(), needs.end());  int i=0;  for(; i<needs.size(); ++i)  {  clone[i] -= offer[i];  if(clone[i] < 0) break;  }  if(i==needs.size())  res = min(res, offer[i]+shoppingOffers(price, special, clone));  }  return res;  }  };  C++2  public:  int sumPrice(vector<int>& price, vector<int>& needs)  {  int res = 0;  for(int i=0; i<price.size(); ++i)  res += price[i]\*needs[i];  return res;  }    int shopping(vector<int>& price, vector<vector<int>>& special, vector<int>& needs, map<vector<int>,int>& mem)  {  if(mem[needs]>0) return mem[needs];    int res = sumPrice(price, needs);  for(auto offer : special)  {  vector<int> clone(needs.begin(), needs.end());  int i=0;  for(; i<needs.size(); ++i)  {  clone[i] -= offer[i];  if(clone[i] < 0) break;  }  if(i==needs.size())  res = min(res, offer[i]+shopping(price, special, clone, mem));  }  mem[needs] = res;  return res;  }    int shoppingOffers(vector<int>& price, vector<vector<int>>& special, vector<int>& needs) {  map<vector<int>,int> mem;  return shopping( price, special,needs, mem);  }  };  Java1  public class Solution {  public int shoppingOffers(List<Integer> price, List<List<Integer>> special, List<Integer> needs) {  Map<List<Integer>, Integer> mem = new HashMap<>();  return shopping(price, special,needs, mem);  }    private int sumPrice(List<Integer> price, List<Integer> needs)  {  int res = 0;  for(int i=0; i<price.size(); ++i)  res += price.get(i) \* needs.get(i);  return res;  }    int shopping(List<Integer> price, List<List<Integer>> special, List<Integer> needs, Map<List<Integer>, Integer> mem)  {  if(mem.containsKey(needs)) return mem.get(needs);    int res = sumPrice(price, needs);  for(List<Integer> offer : special)  {  List<Integer> clone = new ArrayList<Integer>(needs);  int i=0;  for(; i<needs.size(); ++i)  {  clone.set(i, clone.get(i) - offer.get(i));  if(clone.get(i) < 0) break;  }  if(i==needs.size())  res = Math.min(res, offer.get(i)+shopping(price, special, clone, mem));  }  mem.put(needs, res);  return res;  }  } |
|  |

## 排序(sort)

|  |
| --- |
| 386. 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.  [Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question.  C++1 (205ms)  vector<int> lexicalOrder(int n) {  vector<int> res(n);  int temp = 1;  for(int i=0; i<n; ++i)  {  res[i] = temp;  if(temp \* 10 <= n)  temp \*= 10;  else  {  if(temp >= n)  temp /= 10;  temp++;  while(temp%10 == 0)  temp /= 10;  }  }  return res;  }  C++2 (252ms)  vector<int> lexicalOrder(int n) {  vector<int> res(n);  int temp = 1;  for(int i=0; i<n; ++i)  {  res[i] = temp;  if(temp\*10 <= n)  temp \*= 10;  else if(temp%10 != 9 && temp+1<=n)  temp++;  else  {  while((temp/10)%10 == 9)  temp /= 10;  temp = temp/10 + 1;  }  }  return res;  } |
| 506. 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 to 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.   C++1 (12ms) O(nlog(n))  class Solution {  public:  static bool myComp(pair<int,int> a, pair<int, int> b)  {  return a.first > b.first;  }  vector<string> findRelativeRanks(vector<int>& nums) {  int n = nums.size();  vector<string> res(n);  vector<pair<int, int>> rank(n);  for(int i=0; i<n; ++i)  rank[i] = make\_pair(nums[i], i);  sort(rank.begin(), rank.end(), myComp);  if(n>0) res[rank[0].second] = "Gold Medal";  if(n>1) res[rank[1].second] = "Silver Medal";  if(n>2) res[rank[2].second] = "Bronze Medal";  for(int i=n-1; i>=3; --i)  res[rank[i].second] = to\_string(i+1);  return res;  }  }; |
| 455. 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 gi, which is the minimum size of a cookie that the child will be content with; and each cookie j has a size sj. If sj >= gi, 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.  C++1 (79ms)  int findContentChildren(vector<int>& g, vector<int>& s) {  int res = 0, gLen=g.size(), sLen=s.size();  sort(g.begin(), g.end());  sort(s.begin(), s.end());  for(int si=0; res<gLen && si<sLen; ++si)  if(s[si] >= g[res])  res++;  return res;  } |
| Codility [Triangle](https://codility.com/demo/results/trainingMENDRS-A5R/)  A zero-indexed array A consisting of N integers is given. A triplet (P, Q, R) is *triangular* if 0 ≤ P < Q < R < N and:   * A[P] + A[Q] > A[R], * A[Q] + A[R] > A[P], * A[R] + A[P] > A[Q].   For example, consider array A such that:  A[0] = 10 A[1] = 2 A[2] = 5 A[3] = 1 A[4] = 8 A[5] = 20  Triplet (0, 2, 4) is triangular.  Write a function:  int solution(vector<int> &A);  that, given a zero-indexed array A consisting of N integers, returns 1 if there exists a triangular triplet for this array and returns 0 otherwise.  For example, given array A such that:  A[0] = 10 A[1] = 2 A[2] = 5 A[3] = 1 A[4] = 8 A[5] = 20  the function should return 1, as explained above. Given array A such that:  A[0] = 10 A[1] = 50 A[2] = 5 A[3] = 1  the function should return 0.  Assume that:   * N is an integer within the range [0..100,000]; * each element of array A is an integer within the range [−2,147,483,648..2,147,483,647].   Complexity:   * expected worst-case time complexity is O(N\*log(N)); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  sort(A.begin(),A.end());  **for**(**int** i=A.size()-1; i>=2; --i)  {  **if**(A[i]<=0) **return** 0;  **if**((**long**)A[i-1] + (**long**)A[i-2] > A[i]) **return** 1;  }  **return** 0;  }  Testcase  example  example, positive answer, length=6  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  example1  example, answer is zero, length=4  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  extreme\_empty  empty sequence  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  extreme\_single  1-element sequence  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  extreme\_two\_elems  2-element sequence  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  extreme\_negative1  three equal negative numbers  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  extreme\_arith\_overflow1  overflow test, 3 MAXINTs  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  extreme\_arith\_overflow2  overflow test, 10 and 2 MININTs  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  extreme\_arith\_overflow3  overflow test, 0 and 2 MAXINTs  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  medium1  chaotic sequence of values from [0..100K], length=30  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  medium2  chaotic sequence of values from [0..1K], length=50  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  medium3  chaotic sequence of values from [0..1K], length=100  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  large1  chaotic sequence with values from [0..100K], length=10K  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  large2  1 followed by an ascending sequence of ~50K elements from [0..100K], length=~50K  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  large\_random  chaotic sequence of values from [0..1M], length=100K  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  large\_negative  chaotic sequence of negative values from [-1M..-1], length=100K  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  large\_negative2  chaotic sequence of negative values from [-10..-1], length=100K  [▶](https://codility.com/demo/results/trainingMENDRS-A5R/)  large\_negative3  sequence of -1 value, length=100K |
| Codility [CountTriangles](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  A zero-indexed array A consisting of N integers is given. A triplet (P, Q, R) is *triangular* if it is possible to build a triangle with sides of lengths A[P], A[Q] and A[R]. In other words, triplet (P, Q, R) is triangular if 0 ≤ P < Q < R < N and:   * A[P] + A[Q] > A[R], * A[Q] + A[R] > A[P], * A[R] + A[P] > A[Q].   For example, consider array A such that:  A[0] = 10 A[1] = 2 A[2] = 5 A[3] = 1 A[4] = 8 A[5] = 12  There are four triangular triplets that can be constructed from elements of this array, namely (0, 2, 4), (0, 2, 5), (0, 4, 5), and (2, 4, 5).  Write a function:  int solution(vector<int> &A);  that, given a zero-indexed array A consisting of N integers, returns the number of triangular triplets in this array.  For example, given array A such that:  A[0] = 10 A[1] = 2 A[2] = 5 A[3] = 1 A[4] = 8 A[5] = 12  the function should return 4, as explained above.  Assume that:   * N is an integer within the range [0..1,000]; * each element of array A is an integer within the range [1..1,000,000,000].   Complexity:   * expected worst-case time complexity is O(N2); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++ (90%)  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size(), res = 0;  sort(A.begin(), A.end(), greater<**int**>());  **for**(**int** i=0; i<n-2; ++i)  {  **int** minj = n-1;  **for**(**int** j = i+1; j<minj; ++j)  {  **int** k = j+1;  **while**(k<n && A[i] < A[j] + A[k]) ++k;  minj = k;  res += k-j-1;  }  }  **return** res;  }  C++2 (100%)  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** res = 0;  **int** n = A.size();  sort(A.begin(), A.end());  **for**(**int** i=0; i<n-2; ++i)  {  **int** left = i+1;  **int** right = i+2;  **while**(left < n-1)  {  **if**(right<n && A[i]+A[left] > A[right])  right++;  **else**  {  res += right - left -1;  left++;  }  }  }  **return** res;  }  Testcase  extreme\_empty  empty sequence + [5,3,3]  [▶](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  extreme\_single  1-element sequence + [5,3,3]  [▶](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  extreme\_two\_elems  2-element sequence + [5,3,3]  [▶](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  extreme\_arith\_overflow  overflow test, 3 MAXINTs + [5,3,3]  [▶](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  simple  [▶](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  medium1  chaotic sequence of values from [1..100K], length=30  [▶](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  medium2  chaotic sequence of values from [1..1K], length=50  [▶](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  large  chaotic sequence with values from [1..10], length=200  [▶](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  large2  1 followed by an ascending sequence of ~1K elements from [1..2K]  [▶](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  large\_random  chaotic sequence of values from [1..1M], length=1K  [▶](https://codility.com/demo/results/trainingWJKFRY-4KZ/)  large\_the\_same  sequence of the same value value  ✘  TIMEOUT ERROR  running time: 0.17 sec., time limit: 0.10 sec. |
| Codility [**NumberOfDiscIntersections**](https://codility.com/programmers/lessons/6-sorting/number_of_disc_intersections/) We draw N discs on a plane. The discs are numbered from 0 to N − 1. A zero-indexed array A of N non-negative integers, specifying the radiuses of the discs, is given. The J-th disc is drawn with its center at (J, 0) and radius A[J].  We say that the J-th disc and K-th disc intersect if J ≠ K and the J-th and K-th discs have at least one common point (assuming that the discs contain their borders).  The figure below shows discs drawn for N = 6 and A as follows:  A[0] = 1 A[1] = 5 A[2] = 2 A[3] = 1 A[4] = 4 A[5] = 0  https://codility-frontend-prod.s3.amazonaws.com/media/task_static/number_of_disc_intersections/static/images/auto/0eed8918b13a735f4e396c9a87182a38.png  There are eleven (unordered) pairs of discs that intersect, namely:   * discs 1 and 4 intersect, and both intersect with all the other discs; * disc 2 also intersects with discs 0 and 3.   Write a function:  int solution(vector<int> &A);  that, given an array A describing N discs as explained above, returns the number of (unordered) pairs of intersecting discs. The function should return −1 if the number of intersecting pairs exceeds 10,000,000.  Given array A shown above, the function should return 11, as explained above.  Assume that:   * N is an integer within the range [0..100,000]; * each element of array A is an integer within the range [0..2,147,483,647].   Complexity:   * expected worst-case time complexity is O(N\*log(N)); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  int solution(vector<int> &A) {  // write your code in C++14 (g++ 6.2.0)  int n = A.size();  vector<int> low(n);  vector<long> high(n);  for(int i=0; i<n; ++i)  {  low[i] = i - A[i];  high[i] = (long)i + A[i];  }  sort(low.begin(), low.end());  sort(high.begin(), high.end());  int res = 0;  for(int i=0, j=0; i<n && j<n; ++i)  {  while(j<n && high[i] >= low[j])  {  res += j - i;  j++;  }  if(res > 10000000) return -1;  }  return res;  }  Testcase  simple3  empty and [10]  small3  overflow  arithmetic overflow tests  medium1  medium2  medium3  medium4  10M\_intersections  10.000.000 intersections  **WRONG ANSWER**  got 19773477 expected -1  **WRONG ANSWER**  got 49523599 expected -1  big3  [0]\*50000 |
| Codility [MinAbsSumOfTwo](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  Let A be a non-empty zero-indexed array consisting of N integers.  The *abs sum of two* for a pair of indices (P, Q) is the absolute value |A[P] + A[Q]|, for 0 ≤ P ≤ Q < N.  For example, the following array A:  A[0] = 1 A[1] = 4 A[2] = -3  has pairs of indices (0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2).  The abs sum of two for the pair (0, 0) is A[0] + A[0] = |1 + 1| = 2.  The abs sum of two for the pair (0, 1) is A[0] + A[1] = |1 + 4| = 5.  The abs sum of two for the pair (0, 2) is A[0] + A[2] = |1 + (−3)| = 2.  The abs sum of two for the pair (1, 1) is A[1] + A[1] = |4 + 4| = 8.  The abs sum of two for the pair (1, 2) is A[1] + A[2] = |4 + (−3)| = 1.  The abs sum of two for the pair (2, 2) is A[2] + A[2] = |(−3) + (−3)| = 6.  Write a function:  int solution(vector<int> &A);  that, given a non-empty zero-indexed array A consisting of N integers, returns the minimal abs sum of two for any pair of indices in this array.  For example, given the following array A:  A[0] = 1 A[1] = 4 A[2] = -3  the function should return 1, as explained above.  Given array A:  A[0] = -8 A[1] = 4 A[2] = 5 A[3] =-10 A[4] = 3  the function should return |(−8) + 5| = 3.  Assume that:   * N is an integer within the range [1..100,000]; * each element of array A is an integer within the range [−1,000,000,000..1,000,000,000].   Complexity:   * expected worst-case time complexity is O(N\*log(N)); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  sort(A.begin(), A.end());  **int** minAbs = 2e9;  **int** i = 0, j=A.size()-1;  **while**(i<=j)  {  minAbs = min(minAbs, min(**abs**(A[i]+A[j]), min(**abs**(A[i]+A[i]),**abs**(A[j]+A[j]))));  **if**(i+1>=j || minAbs == 0) **return** minAbs;  **int** a = **abs**(A[i] + A[j-1]);  **int** b = **abs**(A[j] + A[i+1]);  **if**(a<=b) j--;  **else** i++;  }  **return** minAbs;  }  Testcase  extreme\_single  sequences of 1 elements  [▶](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  extreme\_double  sequences of 2 elements  [▶](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  positive\_small  only positive numbers  [▶](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  negative\_small  only negative numbers  [▶](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  random\_small  random sequence, length = ~1000  [▶](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  random\_medium  random sequence, length = ~10,000  [▶](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  arithmetic\_medium  arithemtic sequence, length = ~10,000  [▶](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  random\_large  random sequence, length = ~100,000  [▶](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  extreme\_large  sequence of MAX\_INT, length = ~100,000  [▶](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  arithmetic\_large  arithmetic sequence, length = ~100,000  [▶](https://codility.com/demo/results/trainingD3B5P5-WAZ/)  constant\_distance  constant distance between all elements, length = 100,000 |
|  |

### 插入排序(Insert)

#### 直接插入排序(Insert Sort) O(n^2)

|  |
| --- |
| 147. Insertion Sort List  Sort a linked list using insertion sort.  C++1  ListNode\* insertionSortList(ListNode\* head) {  ListNode dummy(0);  //dummy->next = head; //if has this connection, will create a circle, will be ETL  ListNode\* cur=head, \*pre = &dummy;  while(cur != NULL)  {  ListNode\* temp = cur->next;  if(pre->next==NULL || pre->next->val > cur->val) pre = &dummy;  while(pre->next!=NULL && pre->next->val < cur->val)  pre = pre->next;  cur->next = pre->next;  pre->next = cur;  cur = temp;  }  return dummy.next;  }  C++2  ListNode\* insertionSortList(ListNode\* head) {  if(head==NULL || head->next==NULL) return head;  ListNode\* newHead = new ListNode(0);  ListNode\* cur = head, \*pre = newHead, \*move=NULL;  while(cur!=NULL)  {  move = cur->next;  if(!pre || !pre->next || pre->next->val >= cur->val) pre = newHead;  while(pre->next!=NULL && pre->next->val<cur->val)  pre = pre->next;  cur->next = pre->next;  pre->next = cur;  // pre = newHead;  cur = move;  }  return newHead->next;  }  C++3 (traditional way, very slow)  ListNode\* insertionSortList(ListNode\* head) {  if(head==NULL || head->next==NULL) return head;  ListNode\* newHead = new ListNode(0);  ListNode\* cur = head, \*pre = newHead, \*move=NULL;  while(cur!=NULL)  {  move = cur->next;  while(pre->next!=NULL && pre->next->val<cur->val)  pre = pre->next;  cur->next = pre->next;  pre->next = cur;  pre = newHead;  cur = move;  }  return newHead->next;  } |
|  |
|  |

#### 折半插入排序(Binary Insert Sort)

#### 希尔排序(Shell Sort)

### 交换排序(Exchange)

冒泡排序(Bubble Sort) O(n^2)

快速排序(Quick Sort)?? O(nlogn)  
1003. Mixing milk

### 选择排序(Select)

直接选择排序(Select Sort) O(n^2)  
锦标赛排序(Tournament Sort) O(nlogn)  
堆排序(Heap Sort) O(nlogn)

### 归并排序(Merge)

|  |
| --- |
| 148. Sort List  Sort a linked list in *O*(*n* log *n*) time using constant space complexity.  C++1 (TLE) quick sort  class Solution {  public:  ListNode\* merge(ListNode\* left, ListNode\* mid, ListNode\* right)  {  mid->next = right;  if(left==NULL) return mid;    ListNode dummy(0);  dummy.next = left;  while(left->next!=NULL) left = left->next;  left->next = mid;  return dummy.next;  }  ListNode\* sortList(ListNode\* head) {  if(head == NULL || head->next==NULL) return head;  ListNode left(0), right(0);  ListNode \*L=&left, \*R = &right, \*cur=head->next;  while(cur != NULL)  {  if(cur->val <= head->val)  {  L->next = cur;  L = L->next;  }  else  {  R->next = cur;  R = R->next;  }  cur = cur->next;  }  L->next = NULL;  R->next = NULL;  return merge(sortList(left.next), head, sortList(right.next));  }  };  C++2 Divide and conquer  class Solution {  public:  ListNode\* merge(ListNode\* left, ListNode\* right)  {  if(left==NULL) return right;  if(right==NULL) return left;  if(left->val < right->val)  {  left->next = merge(left->next, right);  return left;  }  else  {  right->next = merge(left, right->next);  return right;  }  }  ListNode\* sortList(ListNode\* head) {  if(head==NULL || head->next==NULL) return head;  ListNode \*pre, \*slow=head, \*fast=head;  while(fast!=NULL && fast->next!=NULL)  {  pre = slow;  slow = slow->next;  fast = fast->next->next;  }  pre->next = NULL;  return merge(sortList(head), sortList(slow));  }  }; |
|  |
|  |

### 基数排序(Radix)

### 桶排序 (Bucket Sort)

|  |
| --- |
| 347. 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 ≤ *k* ≤ number of unique elements. * Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size.   C++1 (19ms)  vector<int> topKFrequent(vector<int>& nums, int k) {  unordered\_map<int, int> count;  for(auto num : nums)  count[num]++;  priority\_queue<int, vector<int>, greater<int>> pq;  for(auto ele : count)  {  pq.push(ele.second);  if(pq.size()>k) pq.pop();  }  vector<int>res;  for(auto ele : count)  if(ele.second >= pq.top())  res.push\_back(ele.first);  return res;  }  C++2  vector<int> topKFrequent(vector<int>& nums, int k) {  unordered\_map<int, int> counts;  for(int n : nums) ++counts[n];  vector<vector<int>> buckets(nums.size()+1); // no 0 frequent elements  for(auto m : counts)  {  buckets[m.second].push\_back(m.first);  }  vector<int> res;  for(int i=buckets.size()-1; i>=0; --i)  {  for(int j=0; j<buckets[i].size(); ++j)  {  res.push\_back(buckets[i][j]);  if(res.size()>=k) return res;  }  }  return res;  }  Java1  public List<Integer> topKFrequent(int[] nums, int k) {  Map<Integer, Integer> freqMap = new HashMap<>();  PriorityQueue<Map.Entry<Integer, Integer>> pq =  new PriorityQueue<>((a, b) -> a.getValue() - b.getValue());  List<Integer> res = new ArrayList<>();    for(int num : nums)  {  int count = freqMap.getOrDefault(num, 0);  freqMap.put(num, count + 1);  }    for(Map.Entry<Integer, Integer> map : freqMap.entrySet())  {  pq.offer(map);  if(pq.size()>k)  pq.poll();  }    while(!pq.isEmpty())  {  int num = pq.poll().getKey();  res.add(num);  }  return res;  }  Java2  public List<Integer> topKFrequent(int[] nums, int k) {  int len = nums.length;  Map<Integer, Integer> freqMap = new HashMap<>();  List<Integer> [] buckets = new List[len+1];  List<Integer> res = new ArrayList<>();    for(int num : nums)  {  freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);  }    for(int key : freqMap.keySet())  {  int freq = freqMap.get(key);  if(buckets[freq] == null)  {  buckets[freq] = new ArrayList();  }  buckets[freq].add(key);  }  /\*  for(Map.Entry<Integer, Integer> map : freqMap.entrySet())  {  int freq = map.getValue();  if(buckets[freq] == null)  {  buckets[freq] = new ArrayList();  }  buckets[freq].add(map.getKey());  }  \*/    for(int i=len; i>0; --i)  {  if(buckets[i]==null) continue;  for(int j=0; j<buckets[i].size(); ++j)  {  res.add(buckets[i].get(j));  if(res.size()>= k)  return res;  }  }  return res;  } |
| Codility [PrefixSet](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  A non-empty zero-indexed array A consisting of N integers is given. The *first covering prefix* of array A is the smallest integer P such that 0≤P<N and such that every value that occurs in array A also occurs in sequence A[0], A[1], ..., A[P].  For example, the first covering prefix of the following 5−element array A:  A[0] = 2 A[1] = 2 A[2] = 1 A[3] = 0 A[4] = 1  is 3, because sequence [ A[0], A[1], A[2], A[3] ] equal to [2, 2, 1, 0], contains all values that occur in array A.  Write a function  int solution(vector<int> &A);  that, given a zero-indexed non-empty array A consisting of N integers, returns the first covering prefix of A.  For example, given array A such that  A[0] = 2 A[1] = 2 A[2] = 1 A[3] = 0 A[4] = 1  the function should return 3, as explained above.  Assume that:   * N is an integer within the range [1..1,000,000]; * each element of array A is an integer within the range [0..N−1].   Complexity:   * expected worst-case time complexity is O(N); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** n = A.size();  **vector**<**bool**> occurs(n);  **int** count = 0;  **for**(**int** a : A)  **if**(occurs[a] == **false**)  {  occurs[a] = **true**;  count++;  }  **for**(**int** i=0; i<n; ++i)  **if**(occurs[A[i]] == **true**)  {  occurs[A[i]] = **false**;  **if**(--count==0) **return** i;  }  **return** n-1;  }  Testcase  extreme\_single  1-element sequence  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  extreme\_two  1-element sequence  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  extreme\_constant  constant sequence  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  extreme\_identity  identity permutation  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  extreme\_permutation  permutation  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  simple1  very simple sequence  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  simple2  simple sequence  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  binary  binary sequence  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  periodic  periodic pattern  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_dec\_100  random test 100 elements, 37 different values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_dec\_1000  random test 1000 elements, 34 different values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_dec\_10000  random test 10 000 elements, 30 different values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_dec\_100000  random test 100 000 elements, 27 different values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_sqrt\_100  random test 100 elements, 10 different values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_sqrt\_1000  random test 1000 elements, 31 different values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_sqrt\_10000  random test 10 000 elements, 100 different values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_sqrt\_100000  random test 100 000 elements, 316 different values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_n\_log\_100  random test 100 elements and n/log\_2 n values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_n\_log\_1000  random test 1000 elements and n/log\_2 n values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_n\_log\_10000  random test 10 000 elements and n/log\_2 n values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_n\_log\_100000  random test 100 000 elements and n/log\_2 n values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_n\_100  random test 100 elements and values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_n\_1000  random test 1000 elements and values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_n\_10000  random test 10 000 elements and values.  [▶](https://codility.com/demo/results/trainingF2JCMQ-2S2/)  random\_n\_100000  random test 100 000 elements and values. |
| 274. 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](https://en.wikipedia.org/wiki/H-index): "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.  C++1  int hIndex(vector<int>& citations) {  sort(citations.begin(), citations.end());  int n = citations.size();  int i=0, j=n-1;  while(i<=j)  {  int mid = i + (j-i)/2;  if(citations[mid] == n-mid) return n-mid;  else if(citations[mid] > n-mid)  j = mid-1;  else  i = mid+1;  }  return n-i;  }  C++2  int hIndex(vector<int>& citations) {  sort(citations.begin(), citations.end());  int len = citations.size();  for(int i=0; i<len; ++i)  {  if(len-i <= citations[i])  return len-i;  }  return 0;  }  Java1 (Bucket Sort)  public int hIndex(int[] citations) {  int len = citations.length;  int[] count = new int[len+1];  for(int cit : citations){  if(cit >= len)  count[len]++;  else  count[cit]++;  }    int citSum = 0;  for(int i=len; i>0; --i){  citSum += count[i]; //number of papers, not num of citations  if(citSum >= i)  return i;  }  return 0;  } |
|  |

## 查找(Search)

### 二分(Binary Search)

|  |
| --- |
| 374. 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.  C++ (3ms) (O(log(n)))  int guessNumber(int n) {  int i=1, j=n;  while(i < j)  {  int mid = i + (j-i)/2;  int res = guess(mid);  if(res==0) return mid;  else if(res == -1) j=mid-1;  else i=mid+1;  }  return i;  } |
| 375. 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 at least you need to have to guarantee a **win**.  C++1  class Solution {  public:  int MinToWin(vector<vector<int>>& dp, int start, int end)  {  if(start >= end) return 0;  if(dp[start][end]!=0) return dp[start][end];  int minV = INT\_MAX;  for(int i=start; i<end; ++i)  {  int maxLoc = i + max(MinToWin(dp, start, i-1), MinToWin(dp, i+1,end));  minV = min(minV, maxLoc);  }  dp[start][end]= minV;  return dp[start][end];  }  int getMoneyAmount(int n) {  vector<vector<int>> dp(n+1, vector<int>(n+1));  return MinToWin(dp, 1, n);  }  }; |
| 278. 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.  // Forward declaration of isBadVersion API.  bool isBadVersion(int version);  C++1 (0ms) O(log(n))  class Solution {  public:  int firstBadVersion(int n) {  int i=1, j=n;  while(i<j)  {  int mid = i + (j-i)/2;  if(isBadVersion(mid)) j=mid;  else i = mid+1;  }  return i;  }  };  C++2 (0ms) O(log(n))  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;  } |
| 378. 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 ≤ k ≤ n2.  C++1 (39ms) O(nlog(n)2)  public:  int NumLessThan(vector<int>& row, int key)  {  int i=0, j=row.size();  while(i<j)  {  int mid = i + (j-i)/2;  if(row[mid]<=key) i = mid+1;  else j=mid;  }  return i;  }  int kthSmallest(vector<vector<int>>& matrix, int k) {  int n = matrix.size();  int left = matrix[0][0], right = matrix[n-1][n-1];  while(left < right)  {  int mid = left + (right-left)/2;  int count = 0;  for(auto row : matrix) //for(int i=0; i<n && matrix[i][0]<=mid; ++i)  count += NumLessThan(row, mid);  //int count = upper\_bound(arow.begin(), arow.end(), mid) - arow.begin();  if(count < k) left = mid + 1;  else right = mid;  }  return left;  }  };  Java1  public class Solution {  public int kthSmallest(int[][] matrix, int k) {  int n = matrix.length;  int left = matrix[0][0], right = matrix[n-1][n-1];  while(left < right)  {  int mid = left + (right-left)/2;  int count = 0;  for(int i=0; i<n && matrix[i][0]<=mid; ++i)  {  count += countLess(matrix[i], mid);  }  if(count < k) left = mid+1;  else right = mid;  }  return left;  }    private int countLess(int[] arow, int target)  {  int left = 0, right = arow.length;  while(left < right)  {  int mid = left + (right-left)/2;  if(arow[mid] <= target) left = mid + 1;  else right = mid;  }  return left;  }  } |
| 153. Find Minimum in Rotated Sorted Array  Suppose an array sorted in ascending order 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.  C++1 (3ms) O(log(n))  int findMin(vector<int>& nums) {  int i=0, j=nums.size()-1;  while(i<j)  {  int mid = i + (j-i)/2;  if(nums[mid]<nums[i] || nums[mid]<nums[j]) j = mid;  else i = mid+1;  }  return nums[i];  }  C++2  int findMin(vector<int>& nums) {  int i=0, j=nums.size()-1, mid;  while(i<=j)  {  if(i==j) return nums[i];  mid = (i+j)/2;  if(nums[j]<=nums[mid]) i=mid+1;  else j=mid;  }  return 0;  }  C++3  int findMin(vector<int>& nums) {  int i=0, j=nums.size()-1, mid;  while(i<=j)  {  if(nums[i]<=nums[j]) return nums[i];  mid = (i+j)/2;  if(nums[j]<=nums[mid]) i=mid+1;  else j=mid;  }  return 0;  }  Java1  public int findMin(int[] nums) {  int left=0, right=nums.length-1;  while(left < right)  {  int mid = left + (right-left)/2;  if(nums[mid] > nums[right])  left = mid + 1;  else  right = mid;  }  return nums[left];  } |
| 35. 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 → 2 [1,3,5,6], 2 → 1 [1,3,5,6], 7 → 4 [1,3,5,6], 0 → 0  C++1 (8ms)  int searchInsert(vector<int>& nums, int target) {  int i =0, j=nums.size(), mid;  while(i<j)  {  mid = i + (j-i)/2; // prevent overflow  if(nums[mid]>=target) j = mid;  else i = mid+1;  }  return i;  }  C++2 (16ms)  int searchInsert(vector<int>& nums, int target) {  return lower\_bound(nums.begin(), nums.end(), target) - nums.begin();  }  Java1  public int searchInsert(int[] nums, int target) {  int lo=0, hi=nums.length-1;  while(lo <= hi)  {  int mid = lo + (hi-lo)/2;  if(target == nums[mid])  return mid;  else if(target > nums[mid])  lo = mid + 1;  else  hi = mid - 1;  }  return lo;  } |
| Codility [NailingPlanks](https://codility.com/demo/results/trainingFZPBK3-63N/)  You are given two non-empty zero-indexed arrays A and B consisting of N integers. These arrays represent N planks. More precisely, A[K] is the start and B[K] the end of the K−th plank.  Next, you are given a non-empty zero-indexed array C consisting of M integers. This array represents M nails. More precisely, C[I] is the position where you can hammer in the I−th nail.  We say that a plank (A[K], B[K]) is nailed if there exists a nail C[I] such that A[K] ≤ C[I] ≤ B[K].  The goal is to find the minimum number of nails that must be used until all the planks are nailed. In other words, you should find a value J such that all planks will be nailed after using only the first J nails. More precisely, for every plank (A[K], B[K]) such that 0 ≤ K < N, there should exist a nail C[I] such that I < J and A[K] ≤ C[I] ≤ B[K].  For example, given arrays A, B such that:  A[0] = 1 B[0] = 4 A[1] = 4 B[1] = 5 A[2] = 5 B[2] = 9 A[3] = 8 B[3] = 10  four planks are represented: [1, 4], [4, 5], [5, 9] and [8, 10].  Given array C such that:  C[0] = 4 C[1] = 6 C[2] = 7 C[3] = 10 C[4] = 2  if we use the following nails:   * 0, then planks [1, 4] and [4, 5] will both be nailed. * 0, 1, then planks [1, 4], [4, 5] and [5, 9] will be nailed. * 0, 1, 2, then planks [1, 4], [4, 5] and [5, 9] will be nailed. * 0, 1, 2, 3, then all the planks will be nailed.   Thus, four is the minimum number of nails that, used sequentially, allow all the planks to be nailed.  Write a function:  int solution(vector<int> &A, vector<int> &B, vector<int> &C);  that, given two non-empty zero-indexed arrays A and B consisting of N integers and a non-empty zero-indexed array C consisting of M integers, returns the minimum number of nails that, used sequentially, allow all the planks to be nailed.  If it is not possible to nail all the planks, the function should return −1.  For example, given arrays A, B, C such that:  A[0] = 1 B[0] = 4 A[1] = 4 B[1] = 5 A[2] = 5 B[2] = 9 A[3] = 8 B[3] = 10 C[0] = 4 C[1] = 6 C[2] = 7 C[3] = 10 C[4] = 2  the function should return 4, as explained above.  Assume that:   * N and M are integers within the range [1..30,000]; * each element of arrays A, B, C is an integer within the range [1..2\*M]; * A[K] ≤ B[K].   Complexity:   * expected worst-case time complexity is O((N+M)\*log(M)); * expected worst-case space complexity is O(M), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **bool** **canBeNailed**(**vector**<**int**> &A, **vector**<**int**> &B, **vector**<**int**> &C, **int** cSize, **int** max)  {  **vector**<**int**> profixSum(max);  **for**(**int** i= 0; i<cSize; ++i)  profixSum[C[i]]++;  **for**(**int** i= 2; i<max; ++i)  profixSum[i] += profixSum[i-1];    **for**(**unsigned** **int** i=0; i<A.size(); ++i)  **if**(profixSum[A[i]-1] - profixSum[B[i]] == 0)  **return** **false**;  **return** **true**;  }  **int** **solution**(**vector**<**int**> &A, **vector**<**int**> &B, **vector**<**int**> &C) {  // write your code in C++14 (g++ 6.2.0)  **int** i = 0, j = C.size();  **int** max = 2 \* C.size() + 1;  **int** res = -1;  **while**(i <= j)  {  **int** mid = i + (j-i)/2;  **if**(canBeNailed(A, B, C, mid, max))  {  res = mid;  j = mid-1;  }  **else** i = mid + 1;  }  **return** res;  }  Testcase  extreme\_single  single nail and single plank  [▶](https://codility.com/demo/results/trainingFZPBK3-63N/)  extreme\_point  nail is a point [1, 1]  [▶](https://codility.com/demo/results/trainingFZPBK3-63N/)  few\_nails\_in\_the\_same\_place  few nails are in the same place  [▶](https://codility.com/demo/results/trainingFZPBK3-63N/)  random\_small  random sequence, length = ~100  [▶](https://codility.com/demo/results/trainingFZPBK3-63N/)  random\_medium  random sequence, length = ~10,000  [▶](https://codility.com/demo/results/trainingFZPBK3-63N/)  random\_large  random sequence, length = ~30,000  [▶](https://codility.com/demo/results/trainingFZPBK3-63N/)  extreme\_large\_planks  all large planks, length = ~30,000  [▶](https://codility.com/demo/results/trainingFZPBK3-63N/)  large\_point  all planks are points, length = ~30,000 |
| Codility [MinMaxDivision](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  You are given integers K, M and a non-empty zero-indexed array A consisting of N integers. Every element of the array is not greater than M.  You should divide this array into K blocks of consecutive elements. The size of the block is any integer between 0 and N. Every element of the array should belong to some block.  The sum of the block from X to Y equals A[X] + A[X + 1] + ... + A[Y]. The sum of empty block equals 0.  The *large sum* is the maximal sum of any block.  For example, you are given integers K = 3, M = 5 and array A such that:  A[0] = 2 A[1] = 1 A[2] = 5 A[3] = 1 A[4] = 2 A[5] = 2 A[6] = 2  The array can be divided, for example, into the following blocks:   * [2, 1, 5, 1, 2, 2, 2], [], [] with a large sum of 15; * [2], [1, 5, 1, 2], [2, 2] with a large sum of 9; * [2, 1, 5], [], [1, 2, 2, 2] with a large sum of 8; * [2, 1], [5, 1], [2, 2, 2] with a large sum of 6.   The goal is to minimize the large sum. In the above example, 6 is the minimal large sum.  Write a function:  int solution(int K, int M, vector<int> &A);  that, given integers K, M and a non-empty zero-indexed array A consisting of N integers, returns the minimal large sum.  For example, given K = 3, M = 5 and array A such that:  A[0] = 2 A[1] = 1 A[2] = 5 A[3] = 1 A[4] = 2 A[5] = 2 A[6] = 2  the function should return 6, as explained above.  Assume that:   * N and K are integers within the range [1..100,000]; * M is an integer within the range [0..10,000]; * each element of array A is an integer within the range [0..M].   Complexity:   * expected worst-case time complexity is O(N\*log(N+M)); * expected worst-case space complexity is O(1), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1  **int** **solution**(**int** K, **int** M, **vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **int** m = 0;  **for**(**int** a : A)  **if**(a>m) m = a;  **int** n = A.size();  **int** minSum = m, maxSum = n\*m;  **while**(minSum < maxSum)  {  **int** blockSum = minSum + (maxSum-minSum)/2;  **int** block = 0;  **for**(**int** i=0, sum=0; i<n && block<K; ++i)  {  sum += A[i];  **if**(sum > blockSum)  {  sum = A[i];  block++;  }  }    **if**(block != K) maxSum = blockSum;  **else** minSum = blockSum + 1;  }  **return** maxSum;  }  Testcase  extreme\_single  single elements  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  extreme\_double  single and double elements  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  extreme\_min\_max  maximal / minimal values  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  simple1  simple tests  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  simple2  simple tests  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  tiny\_random\_ones  random values {0, 1}, N = 100  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  small\_random\_ones  random values {0, 1}, N = 100  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  medium\_zeros  many zeros and 99 in the middle, length = 15,000  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  medium\_random  random values {1, 100}, N = 20,000  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  large\_random  random values {0, ..., MAX\_INT}, N = 100,000  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  large\_random\_ones  random values {0, 1}, N = 100,000  [▶](https://codility.com/demo/results/trainingPVZCBB-9TZ/)  all\_the\_same  all the same values, N = 100,000 |
| 162. 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.](https://leetcode.com/problems/find-peak-element/)  **Note:**  Your solution should be in logarithmic complexity.  C++1  int findPeakElement(vector<int>& nums) {  int n=nums.size();  if(n==1) return 0;  long left = LONG\_MIN;  for(int i=0; i<n-1; ++i)  {  if(nums[i]>left && nums[i]>nums[i+1]) return i;  left = nums[i];  }  return n-1;  }  C++2  int findPeakElement(vector<int>& nums) {  int len = nums.size()-1;  if(len<=0) return 0;  for(int i=0; i<len; ++i)  {  if(nums[i]>nums[i+1]) return i;  //if(nums[len-i] - nums[len-i-1] > 0) return len-i;  }  return len;  }  C++3  int findPeakElement(vector<int>& nums) {  int j = nums.size()-1;  int i = 0, mid;  while(i<j)  {  mid = i + (j-i)/2;  if(nums[mid]<nums[mid+1]) i = mid+1;  else j = mid;  }  return i;  } |
|  |
| 410. 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:** If *n* is the length of array, assume the following constraints are satisfied:   * 1 ≤ *n* ≤ 1000 * 1 ≤ *m* ≤ min(50, *n*)   **Examples:**  Input:  **nums** = [7,2,5,10,8]  **m** = 2  Output:  18  Explanation:  There are four ways to split **nums** into two subarrays.  The best way is to split it into **[7,2,5]** and **[10,8]**,  where the largest sum among the two subarrays is only 18.  C++1  class Solution {  public:  bool isValidSplit(vector<int>& nums, int m, int target)  {  long sum=0, count=1;  for(int num : nums)  {  sum += num;  if(sum > target)  {  sum = num;  if(++count > m) return false;  }  }  return true;  }  int splitArray(vector<int>& nums, int m) {  int maxV=0, sum=0;  for(int num : nums)  {  maxV = max(maxV, num);  sum += num;  }  if(m==1) return sum;  int i=maxV, j = sum;  while(i<=j)  {  long mid = (j+i)/2;  if(isValidSplit(nums, m, mid))  j = mid-1;  else  i = mid+1;  }  return i;  }  }; |
| 274. 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](https://en.wikipedia.org/wiki/H-index): "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.  C++1  int hIndex(vector<int>& citations) {  sort(citations.begin(), citations.end());  int n = citations.size();  int i=0, j=n-1;  while(i<=j)  {  int mid = i + (j-i)/2;  if(citations[mid] == n-mid) return n-mid;  else if(citations[mid] > n-mid)  j = mid-1;  else  i = mid+1;  }  return n-i;  }  C++2  int hIndex(vector<int>& citations) {  sort(citations.begin(), citations.end());  int len = citations.size();  for(int i=0; i<len; ++i)  {  if(len-i <= citations[i])  return len-i;  }  return 0;  }  Java1 (Bucket Sort)  public int hIndex(int[] citations) {  int len = citations.length;  int[] count = new int[len+1];  for(int cit : citations){  if(cit >= len)  count[len]++;  else  count[cit]++;  }    int citSum = 0;  for(int i=len; i>0; --i){  citSum += count[i]; //number of papers, not num of citations  if(citSum >= i)  return i;  }  return 0;  } |
| 275. H-Index II  **Follow up** for [H-Index](https://leetcode.com/problems/h-index/): What if the citations array is sorted in ascending order? Could you optimize your algorithm?  C++1  int hIndex(vector<int>& citations) {  int n = citations.size();  int i=0, j=n-1;  while(i<=j)  {  int mid = i + (j-i)/2;  if(citations[mid] == n-mid) return n-mid;  else if(citations[mid] > n-mid)  j = mid-1;  else  i = mid+1;  }  return n-i;  }  C++2  int hIndex(vector<int>& citations) {  int len = citations.size();  for(int i=0; i<len; ++i)  {  if(len-i <= citations[i])  return len-i;  }  return 0;  } |
| 81. 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?  Suppose an array sorted in ascending order 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).  Write a function to determine if a given target is in the array.  The array may contain duplicates.  C++1 (6ms) fastest  class Solution {  public:  bool binarySearch(vector<int>& nums, int target, int i, int j)  {  if(i <= j)  {  int mid = i+ (j-i)/2;  if(nums[mid] == target) return true;  else if(nums[i]<=target && target<nums[mid])  return binarySearch(nums, target, i, mid-1);  else if(nums[mid]<target && target<=nums[j])  return binarySearch(nums, target, mid+1, j);  else  return binarySearch(nums, target, i, mid-1) || binarySearch(nums, target, mid+1, j);  }  return false;  }  bool search(vector<int>& nums, int target) {  return binarySearch(nums, target, 0, nums.size()-1);  }  };  C++2  class Solution {  public:  bool binarySearch(vector<int>& nums, int target, int i, int j)  {  if(i > j) return false;  int mid = i+ (j-i)/2;  if(nums[mid] == target) return true;  else if(nums[i]<=target && target<nums[mid])  return binarySearch(nums, target, i, mid-1);  else if(nums[mid]<target && target<=nums[j])  return binarySearch(nums, target, mid+1, j);  else  return binarySearch(nums, target, i, mid-1) || binarySearch(nums, target, mid+1, j);  }  bool search(vector<int>& nums, int target) {  return binarySearch(nums, target, 0, nums.size()-1);  }  };  C++3  bool search(vector<int>& nums, int target) {  int i=0, j = nums.size()-1, mid;  while(i<=j)  {  mid = i + (j-i)/2;  if(target == nums[mid]) return true;  while(i<j && nums[i]==nums[mid] && nums[mid]==nums[j])  {  i++;  j--;  }  if(nums[i] <= nums[mid])  {  if(target>=nums[i] && target<nums[mid]) j = mid-1;  else i = mid+1;  }  else  {  if(target>nums[mid] && target<=nums[j]) i = mid+1;  else j = mid-1;  }  }  return false;  }  C++4  class Solution {  public:  bool Helper(vector<int>& nums, int target, int left, int right)  {  if(left>right) return false;  int mid = left + (right-left)/2;  if(target == nums[mid]) return true;    if(nums[left] < nums[mid])  {  if(target >= nums[left] && target < nums[mid])  return Helper(nums, target, left, mid-1);  else  return Helper(nums, target, mid+1, right);  }  else if(nums[mid] < nums[right])  {  if(target > nums[mid] && target <= nums[right])  return Helper(nums, target, mid+1, right);  else  return Helper(nums, target, left, mid-1);  }  else  {  if(nums[left] == nums[mid] && nums[mid] != nums[right])  {  return Helper(nums, target, mid+1, right);  }  else if(nums[left] != nums[mid] && nums[mid] == nums[right])  {  return Helper(nums, target, left, mid-1);  }  else  {  return Helper(nums, target, left, mid-1) | Helper(nums, target, mid+1, right);  }  }  return false;  }  bool search(vector<int>& nums, int target) {  return Helper(nums, target, 0, nums.size()-1);  }  }; |
| 167. 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 and you may not use the *same* element twice.  **Input:** numbers={2, 7, 11, 15}, target=9 **Output:** index1=1, index2=2  C++1 (6ms)  vector<int> twoSum(vector<int>& numbers, int target) {  vector<int> res(2);  int i=0, j=numbers.size()-1;  while(i<j)  {  int sum = numbers[i] + numbers[j];  if(sum == target)  {  res[0] = i+1;  res[1] = j+1;  return res;  }  else if(sum > target) j--;  else i++;  }  return res;  } |
| 653. Two Sum IV - Input is a BST  Given a Binary Search Tree and a target number, return true if there exist two elements in the BST such that their sum is equal to the given target.  **Example 1:**  **Input:**  5  / \  3 6  / \ \  2 4 7  Target = 9  **Output:** True  **Example 2:**  **Input:**  5  / \  3 6  / \ \  2 4 7  Target = 28  **Output:** False  C++1  bool findTargetIn(unordered\_set<int>& set, TreeNode\* root, int k)  {  if(root == NULL) return false;  if(set.find(k - root->val) != set.end()) return true;  set.insert(root->val);  return findTargetIn(set, root->left, k) || findTargetIn(set, root->right, k);  }    bool findTarget(TreeNode\* root, int k) {  unordered\_set<int> set;  return findTargetIn(set, root, k);  }  Java1  public boolean findTarget(TreeNode root, int k) {  Set<Integer> set = new HashSet<>();  return findTargetIn(set, root, k);  }    private boolean findTargetIn(Set<Integer> set, TreeNode root, int k){  if(root == null) return false;  if(set.contains(k - root.val)) return true;  set.add(root.val);  return findTargetIn(set, root.left, k) || findTargetIn(set, root.right, k);  } |
| 33. Search in Rotated Sorted Array  Suppose an array sorted in ascending order 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.  C++1  int search(vector<int>& nums, int target) {  int i=0, j=nums.size()-1;  while(i<=j)  {  int mid = i + (j-i)/2;  if(nums[mid] == target) return mid;  else if(nums[i]<= nums[mid]) //cause mid is floor int  {  if(nums[i] <= target && target < nums[mid])  j = mid-1;  else  i = mid+1;  }  else if(nums[mid] < target && target<=nums[j])  i = mid+1;  else  j = mid-1;  }  return -1;  }  C++2  int search(vector<int>& nums, int target) {  int i=0, j=nums.size()-1, mid;  while(i<=j)  {  mid = i + (j-i)/2;  if(nums[mid] == target) return mid;  if(nums[mid] < nums[j])  {  if(target>nums[mid] && target<=nums[j])  i = mid+1;  else  j = mid-1;  }  else if(target>=nums[i] && target<nums[mid])  {  j = mid-1;  }  else  i = mid+1;  }  return -1;  } |
| 34. Search for a Range  Given an array of integers sorted in ascending order, 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].  C++1 (O(log(n)))  vector<int> searchRange(vector<int>& nums, int target) {  bool noSuchTarget = true;  int i=0, j=nums.size()-1;  while(i<=j)  {  int mid = i + (j-i)/2;  if(nums[mid] == target)  {  noSuchTarget = false;  break;  }  else if(nums[mid] > target)  j = mid-1;  else  i = mid+1;  }  if(noSuchTarget) return vector<int>(2, -1);  auto lowerB = lower\_bound(nums.begin(), nums.end(), target);  auto higherB = upper\_bound(nums.begin(), nums.end(), target);  vector<int> res(1, lowerB-nums.begin());  res.push\_back(higherB-nums.begin()-1);  return res;  }  C++2 (O(n))  vector<int> searchRange(vector<int>& nums, int target) {  int len = nums.size(), i=0, j=len-1, mid;  vector<int> res(2,-1);  while(i<=j)  {  mid = i + (j-i)/2;  if(target == nums[mid])  {  i=mid;  while(i>0 && nums[i]==nums[i-1]) i--;  while(mid<j && nums[mid]==nums[mid+1]) mid++;  res[0]=i;  res[1]=mid;  return res;  }  else if(target>nums[mid])  {  i = mid+1;  }  else  {  j = mid-1;  }  }  return res;  }  C++3  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 stuck.  }  ret[1] = j;  return ret;  } |
| 483. 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.   Hint  The input can be stored in a long long int, here I use unsigned long long int for a larger range. We need to find k, for 1+k^1+k^2+k^3+...+k^d=n. The smallest possible base is k=2, with has the longest possible representation, i.e., largest d. So, to find the smallest base means to find the longest possible representation "11111....1" based on k. As n<=10^18, so n<(1<<62). We iterate the length of the representation from 62 to 2 (2 can always be valid, with base=n-1), and check whether a given length can be valid.  For a given length d, we use binary search to check whether there is a base k which satisfies 1+k^1+k^2+...k^d=n. The left limit is 1, and the right limit is pow(n,1/d)+1, i.e., the d th square root of n. The code is shown below.  C++1  class Solution {  public:  long binarySearch(long target, int power)  {  long left = 1;  long right = pow(target, 1.0/power)+1;  while(left <= right)  {  long mid = left + (right-left)/2;  long sum = 1, val=1;  for(int i=0; i<power; ++i)  {  val \*= mid;  sum += val;  }  if(sum == target) return mid;  else if(sum > target) right = mid-1;  else left = mid+1;  }  return 0;  }  string smallestGoodBase(string n) {  long target = stol(n);  for(int i=62; i>=1; --i)  {  if(1<<i <target)  {  long res = binarySearch(target, i);  if(res > 0) return to\_string(res);  }  }  return to\_string(target-1);  }  }; |
| 209. Minimum Size Subarray Sum  Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a **contiguous** 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.](https://leetcode.com/problems/minimum-size-subarray-sum/)  **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*).  C++1 O(n\*log(n))  int minSubArrayLen(int s, vector<int>& nums) {  int n = nums.size();  if(n==0 || s==0) return 0;  int res = n+1;  int sum = 0;  for(int i=0; i<n; ++i)  sum[i+1] = sum[i]+nums[i];  for(int i=0; i<n; ++i)  {  int left = i+1;  int right = n;  while(left <= right)  {  int mid = left + (right-left)/2;  if(sum[mid]-sum[i] >= s)  {  res = min(res, mid-i);  right = mid-1;  }  else  left = mid+1;  }  }  return res==n+1 ? 0 : res;  }  C++2 O(n)  int minSubArrayLen(int s, vector<int>& nums) {  int sum =0, start=0, minLen = INT\_MAX;  for(int i=0; i<nums.size(); ++i)  {  sum+=nums[i];  while(sum>=s)  {  minLen = min(minLen, i-start+1);  sum -= nums[start++];  }  }  return minLen==INT\_MAX ? 0:minLen;  } |
| 611. Valid Triangle Number |
|  |

### 树型

#### Binary Tree

|  |
| --- |
| 513. 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**.  C++1 (12ms)  class Solution {  public:  void RescursiveBL(TreeNode\* root, vector<int>& res, int depth)  {  if(root==NULL) return;  if(res.size() == depth) res.push\_back(root->val);  RescursiveBL(root->left, res, depth+1);  RescursiveBL(root->right, res, depth+1);  }  int findBottomLeftValue(TreeNode\* root) {  vector<int> res;  RescursiveBL(root, res, 0);  return res.back();  }  }; |
| 515. 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]  C++1 (12ms)  class Solution {  public:  void RecursiveLV(TreeNode\* root, vector<int>& res, int depth)  {  if(root==NULL) return;  if(depth==res.size()) res.push\_back(root->val);  res[depth] = max(res[depth], root->val);  RecursiveLV(root->left, res, depth+1);  RecursiveLV(root->right, res, depth+1);  }  vector<int> largestValues(TreeNode\* root) {  vector<int>res;  RecursiveLV(root, res, 0);  return res;  }  }; |
| 501. Find Mode in Binary Search Tree  Given a binary search tree (BST) with duplicates, find all the [mode(s)](https://en.wikipedia.org/wiki/Mode_(statistics)) (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).  C++1 (26ms) no meaning for BST, maybe have better methor using BST?  class Solution {  public:  int countMode(TreeNode\* root, unordered\_map<int, int>& count)  {  if(root == NULL) return 0;  count[root->val]++;  return max(count[root->val], max(countMode(root->left, count), countMode(root->right, count)));  }  vector<int> findMode(TreeNode\* root) {  unordered\_map<int, int> count;  int modeSize = countMode(root, count);  vector<int> res;  for(pair<int, int> c : count)  if(c.second == modeSize)  res.push\_back(c.first);  return res;  }  }; |
| 508. 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.  C++1 (16ms)  class Solution {  public:  int CountMaxFreq(TreeNode\* root, unordered\_map<int, int>& count, int& sum)  {  if(root==NULL) return 0;  int maxL = CountMaxFreq(root->left, count, sum);  int sumR = 0;  int maxR = CountMaxFreq(root->right, count, sumR);  sum += root->val + sumR;  count[sum]++;  return max(count[sum], max(maxL, maxR));  }  vector<int> findFrequentTreeSum(TreeNode\* root) {  unordered\_map<int, int> count;  vector<int> res;  int sum = 0;  int mostFreq = CountMaxFreq(root, count, sum);  for(pair<int, int> c : count)  if(c.second == mostFreq) res.push\_back(c.first);  return res;  }  }; |
| 236. 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](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “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.  C++1  class Solution {  public:  TreeNode\* searchNode(TreeNode\* root, TreeNode\* node)  {  if(root==NULL|| root==node) return root;  TreeNode\* left = searchNode(root->left, node);  TreeNode\* right = searchNode(root->right, node);  if(left!=NULL || right != NULL) return node;  return NULL;  }  TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {  if(root==NULL || root==p || root==q) return root;  if(searchNode(root->left, p)!=p && searchNode(root->left, q)!=q)  return lowestCommonAncestor(root->right, p, q);  else if(searchNode(root->right, p)!=p && searchNode(root->right, q)!=q)  return lowestCommonAncestor(root->left, p, q);  return root;  }  };  C++2  TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {  if(root==NULL || root==p || root==q) return root;  TreeNode\* right = lowestCommonAncestor(root->right, p, q);  TreeNode\* left = lowestCommonAncestor(root->left, p, q);  return left&&right ? root: left? left: right;  } |
|  |

#### 二叉搜索树(Binary Search Tree)

|  |
| --- |
| 235. 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](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes v and was the lowest node in T that has both v and was 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.  C++1 (26ms, beats 91.83%)  TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {  if(root==NULL || p==NULL || q==NULL) return NULL;  if((root->val - p->val)\*(root->val - q->val) <= 0) return root;  else if(root->val > p->val)  return lowestCommonAncestor(root->left, p, q);  else  return lowestCommonAncestor(root->right, p, q);  } |
| 230. 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?  **Hint:**   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).   C++1 (12ms)  class Solution {  public:  int countNodes(TreeNode\* root)  {  if(root==NULL) return 0;  return countNodes(root->left) + countNodes(root->right) + 1;  }  int kthSmallest(TreeNode\* root, int k) {  int leftSum = countNodes(root->left) + 1;  if(leftSum == k) return root->val;  else if(leftSum > k) return kthSmallest(root->left, k);  else return kthSmallest(root->right, k-leftSum);  }  };  C++2  class Solution {  public:  int kthSmallestHelper(TreeNode\* root, int& k)  {  if(root==NULL) return -1;  int x = kthSmallestHelper(root->left, k);  if(k==0) return x;  if(--k==0) return root->val;  return kthSmallestHelper(root->right, k);  }  int kthSmallest(TreeNode\* root, int k) {  return kthSmallestHelper(root,k);  }  };  C++3  int kthSmallest(TreeNode\* root, int k) {  stack<TreeNode\*> s;  int count=0;  TreeNode\* t = root;  while(t!=NULL || !s.empty())  {  if(t != NULL)  {  s.push(t);  t = t->left;  }  else  {  t = s.top();  s.pop();  if(++count == k) return t->val;  t = t->right;  }  }  return 0;  }  Java1  public class Solution {  public int kthSmallest(TreeNode root, int k) {  if(root == null) return 0;  int count = countNodes(root.left);  if(count == k-1)  return root.val;  else if(count >= k)  return kthSmallest(root.left, k);  else  return kthSmallest(root.right, k-count-1);  }    private int countNodes(TreeNode root)  {  if(root == null) return 0;  return 1 + countNodes(root.left) + countNodes(root.right);  }  } |
| 530. Minimum Absolute Difference in BST  Given a binary search tree with non-negative values, find the minimum [absolute difference](https://en.wikipedia.org/wiki/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 between 2 and 3).  C++1 (16ms)  class Solution {  public:  int getMaxFromLeftNode(TreeNode\* root)  {  while(root->right!=NULL) root=root->right;  return root->val;  }  int getMinFromRightNode(TreeNode\* root)  {  while(root->left!=NULL) root=root->left;  return root->val;  }  int getMinimumDifference(TreeNode\* root) {  if(root==NULL) return INT\_MAX;  int LD = (root->left!=NULL) ? root->val - getMaxFromLeftNode(root->left) : INT\_MAX;  int RD = (root->right!=NULL) ? getMinFromRightNode(root->right) - root->val : INT\_MAX;  return min(min(LD, RD), min(getMinimumDifference(root->left), getMinimumDifference(root->right)));  }  }; |
| 450. Delete Node in a BST  Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.  Basically, the deletion can be divided into two stages:   1. Search for a node to remove. 2. If the node is found, delete the node.   **Note:** Time complexity should be O(height of tree).  **Example:**  root = [5,3,6,2,4,null,7]  key = 3  5  / \  3 6  / \ \  2 4 7  Given key to delete is 3. So we find the node with value 3 and delete it.  One valid answer is [5,4,6,2,null,null,7], shown in the following BST.  5  / \  4 6  / \  2 7  Another valid answer is [5,2,6,null,4,null,7].  5  / \  2 6  \ \  4 7  C++1 (56ms)  class Solution {  public:  TreeNode\* maxOf(TreeNode\* root)  {  while(root->right != NULL) root = root->right;  return root;  }  TreeNode\* deleteNode(TreeNode\* root, int key) {  if(root==NULL) return NULL;  if(root->val > key)  root->left = deleteNode(root->left, key);  else if(root->val < key)  root->right = deleteNode(root->right, key);  else  {  if(root->left == NULL) return root->right;  if(root->right == NULL) return root->left;  root->val = maxOf(root->left)->val;  root->left = deleteNode(root->left, root->val);  }  return root;  }  };  C++2  class Solution {  public:  int minValOf(TreeNode\* root)  {  while(root->left != NULL) root = root->left;  return root->val;  }  TreeNode\* deleteNode(TreeNode\* root, int key) {  if(root==NULL) return NULL;  if(key < root->val)  root->left = deleteNode(root->left, key);  else if(key > root->val)  root->right = deleteNode(root->right, key);  else  {  if(root->left == NULL)  return root->right;  if(root->right ==NULL)  return root->left;  root->val = minValOf(root->right);  root->right = deleteNode(root->right, root->val);  }  return root;  }  }; |
|  |
|  |

#### 平衡搜索树(AVL Tree)

|  |
| --- |
|  |
|  |
|  |

#### 并查集(Union-Find Set)

#### Trie

|  |
| --- |
| 208. 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.  C++1  class TrieNode{  public:  vector<TrieNode\*> node;  bool isWord;  TrieNode(){  isWord = false;  node = vector<TrieNode\*>(26, nullptr);  }  };  class Trie {  private:  TrieNode\* root;  TrieNode\* find(string& word)  {  TrieNode\* cur = root;  for(int i=0; i<word.length() && cur!=NULL; ++i)  {  cur = cur->node[word[i]-'a'];  }  return cur;  }  public:  /\*\* Initialize your data structure here. \*/  Trie() {  root = new TrieNode();  }    /\*\* Inserts a word into the trie. \*/  void insert(string word) {  TrieNode\* cur = root;  for(char w : word)  {  if(cur->node[w-'a'] == NULL) cur->node[w-'a'] = new TrieNode();  cur=cur->node[w-'a'];  }  cur->isWord = true;  }    /\*\* Returns if the word is in the trie. \*/  bool search(string word) {  TrieNode\* res = find(word);  return res!=NULL && res->isWord;  }    /\*\* Returns if there is any word in the trie that starts with the given prefix. \*/  bool startsWith(string prefix) {  return find(prefix)!=NULL;  }  };  /\*\*  \* Your Trie object will be instantiated and called as such:  \* Trie obj = new Trie();  \* obj.insert(word);  \* bool param\_2 = obj.search(word);  \* bool param\_3 = obj.startsWith(prefix);  \*/ |
| 211. 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.](https://leetcode.com/problems/add-and-search-word-data-structure-design/)  You should be familiar with how a Trie works. If not, please work on this problem: [Implement Trie (Prefix Tree)](https://leetcode.com/problems/implement-trie-prefix-tree/) first.  C++1  class Trie{  public:  vector<Trie\*> children;  bool isWord;  Trie()  {  children = vector<Trie\*>(26);  isWord=false;  }  };  class WordDictionary {  private:  Trie\* root;  public:  /\*\* Initialize your data structure here. \*/  WordDictionary() {  root = new Trie();  }    bool checkAllChriden(string& word, Trie\* root, int star)  {  for(auto child : root->children)  if(child!=NULL)  if(searchWordFromTrie(word, child, star)) return true;  return false;  }  bool searchWordFromTrie(string& word, Trie\* root, int start)  {  if(start==word.length()) return root->isWord;  for(int i=start; i<word.length(); ++i)  {  if(word[i]=='.') return checkAllChriden(word, root, i+1);  else if(root->children[word[i]-'a'] == NULL) return false;  else return searchWordFromTrie(word, root->children[word[i]-'a'], i+1);  }  return false;  }    /\*\* Adds a word into the data structure. \*/  void addWord(string word) {  Trie\* cur = root;  for(char w : word)  {  if(cur->children[w-'a'] == NULL)  cur->children[w-'a'] = new Trie();  cur = cur->children[w-'a'];  }  cur->isWord = 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 searchWordFromTrie(word, root, 0);  }  };  /\*\*  \* Your WordDictionary object will be instantiated and called as such:  \* WordDictionary obj = new WordDictionary();  \* obj.addWord(word);  \* bool param\_2 = obj.search(word);  \*/  C++2  class WordDictionary {  public:  struct Trie{  vector<Trie\*> child;  bool isWord;  Trie(): isWord(false), child(vector<Trie\*>(26, nullptr)){}  };  Trie\* root;  WordDictionary()  {  root = new Trie();  }  // Adds a word into the data structure.  void addWord(string word) {  int len = word.size();  Trie\* cur = root;  for(int i=0; i<len; ++i)  {  int index = word[i]-'a';  if(cur->child[index]==NULL) cur->child[index] = new Trie();  cur = cur->child[index];  }  cur->isWord = true;  }  bool search(const char \* c, Trie\* cur)  {  if(cur==NULL) return false;  if(\*c == '\0') return cur->isWord;  if(\*c == '.')  {  for(int i=0; i<26; ++i)  {  if(search(c+1, cur->child[i])) return true;  }  return false;  }  else  return search(c+1, cur->child[\*c-'a']);    }  // 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 search(word.c\_str(), root);  }  }; |
|  |

### 哈希(Hashing)

|  |
| --- |
| 1. Two Sum   Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.  You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.  **Example:**  Given nums = [2, 7, 11, 15], target = 9,  Because nums[**0**] + nums[**1**] = 2 + 7 = 9,  return [**0**, **1**].  C++1 (9ms) O(n)  vector<int> twoSum(vector<int>& nums, int target) {  vector<int> res(2,0); //vector<int> res(2);  unordered\_map<int, int> map;  for(int i=0; i<nums.size(); ++i)  {  if(map[target-nums[i]]>0)  {  res[0] = map[target-nums[i]]-1;  res[1] = i;  return res;  }  map[nums[i]] = i+1;  }  return res;  }  C++2 (13ms) O(n)  vector<int> twoSum(vector<int>& nums, int target) {  vector<int>res(2,0);  unordered\_map<int, int>map;  for(int i=0; i<nums.size(); ++i)  map[nums[i]] = i+1;  for(int i=0; i<nums.size(); ++i)  {  int ans = target - nums[i];  if(map[ans] > 0 && map[ans]!=i+1)  {  res[0] = map[ans]-1;  res[1] = i;  return res;  }  }  return res;  }  Java1 (6ms) O(n)  public int[] twoSum(int[] nums, int target) {  Map<Integer,Integer> map = new HashMap<>();  for(int i=0; i<nums.length; ++i)  {  int ans = target - nums[i];  if(map.containsKey(ans))  return new int[] {map.get(ans), i};  else  map.put(nums[i], i);  }  return new int[] {0, 0};  } |
| 167. 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 and you may not use the *same* element twice.  **Input:** numbers={2, 7, 11, 15}, target=9 **Output:** index1=1, index2=2  C++1 (6ms)  vector<int> twoSum(vector<int>& numbers, int target) {  vector<int> res(2);  int i=0, j=numbers.size()-1;  while(i<j)  {  int sum = numbers[i] + numbers[j];  if(sum == target)  {  res[0] = i+1;  res[1] = j+1;  return res;  }  else if(sum > target) j--;  else i++;  }  return res;  } |
| 653. Two Sum IV - Input is a BST  Given a Binary Search Tree and a target number, return true if there exist two elements in the BST such that their sum is equal to the given target.  **Example 1:**  **Input:**  5  / \  3 6  / \ \  2 4 7  Target = 9  **Output:** True  **Example 2:**  **Input:**  5  / \  3 6  / \ \  2 4 7  Target = 28  **Output:** False  C++1  bool findTargetIn(unordered\_set<int>& set, TreeNode\* root, int k)  {  if(root == NULL) return false;  if(set.find(k - root->val) != set.end()) return true;  set.insert(root->val);  return findTargetIn(set, root->left, k) || findTargetIn(set, root->right, k);  }    bool findTarget(TreeNode\* root, int k) {  unordered\_set<int> set;  return findTargetIn(set, root, k);  }  Java1  public boolean findTarget(TreeNode root, int k) {  Set<Integer> set = new HashSet<>();  return findTargetIn(set, root, k);  }    private boolean findTargetIn(Set<Integer> set, TreeNode root, int k){  if(root == null) return false;  if(set.contains(k - root.val)) return true;  set.add(root.val);  return findTargetIn(set, root.left, k) || findTargetIn(set, root.right, k);  } |
| 15. 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:** 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]  ]  C++1  vector<vector<int>> threeSum(vector<int>& nums) {  int n = nums.size();  vector<vector<int>> res;    if(n<3) return res;  vector<int> triplet(3);  sort(nums.begin(), nums.end());    for(int i1=0; i1<n-2; ++i1)  {  while(i1>0 && i1<n-2 && nums[i1]==triplet[0]) i1++;  triplet[0] = nums[i1];  int i=i1+1, j=n-1;  while(i<j)  {  int sum = nums[i] + nums[j];  if(sum == -triplet[0])  {  triplet[1] = nums[i++];  triplet[2] = nums[j--];  res.push\_back(triplet);  while(i<j && nums[i]==triplet[1]) i++;  while(j>i && nums[j]==triplet[2]) j--;  }  else if(sum < -triplet[0]) i++;  else j--;  }  }  return res;  }  C++2  vector<vector<int>> threeSum(vector<int>& nums) {  vector<vector<int>> res;  vector<int> arow(3);  sort(nums.begin(), nums.end());  int sum, left, right, target,n = nums.size();  for(int i=0; i<n-2; ++i)  {  while(i>0 && i<n-2 && nums[i]==arow[0]) i++;  target = -nums[i];  if(target<0) return res;  arow[0] = nums[i];  left = i+1;  right = n-1;  while(left<right)  {  sum = nums[left] + nums[right];  arow[1] = nums[left];  arow[2] = nums[right];  if(sum==target)  {  res.push\_back(arow);  while(left<right && nums[++left]==arow[1]);  while(left<right && nums[--right]==arow[2]);  }  else if(sum<target)  {  while(left<right && nums[++left]==arow[1]);  }  else  {  while(left<right && nums[--right]==arow[2]);  }  }  }  return res;  } |
| 16. 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).  C++1  int threeSumClosest(vector<int>& nums, int target) {  sort(nums.begin(), nums.end());  int closest = nums[0] + nums[1] + nums[2];  int n = nums.size();  for(int k=0; k<n-2; ++k)  {  int i=k+1, j=n-1;  while(i<j)  {  int x = nums[k] + nums[i] + nums[j];  if(abs(x-target) < abs(closest-target))  {  closest = x;  if(x == target) return target;  }  if(x > target) j--;  else i++;  }  }  return closest;  } |
| 18. 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:** 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]  ]  C++1  vector<vector<int>> fourSum(vector<int>& nums, int target) {  vector<vector<int>> res;  int size= nums.size();  if(size < 4) return res;  sort(nums.begin(), nums.end());  int aRow[4] = {nums[0], nums[1],nums[2],nums[3]};  for(int i1=0; i1<size-3;++i1)  {  while(i1!=0 && i1<size-3 && nums[i1]==aRow[0]) i1++;  aRow[0] = nums[i1];  aRow[1] = nums[i1+1];  for(int i2=i1+1; i2<size-2; ++i2)  {  while(i2!=i1+1 && i2<size-2 && nums[i2]==aRow[1]) i2++;  aRow[1] = nums[i2];  int i3=i2+1, i4=size-1;  while(i3<i4)  {  int sum = nums[i1] + nums[i2] +nums[i3] + nums[i4];  if(sum == target)  {  aRow[2] = nums[i3];  aRow[3] = nums[i4];  res.push\_back(vector<int>(aRow, aRow+4));  while(i3<size-1 && nums[i3]==aRow[2]) i3++;  while(i4>i2+1 && nums[i4]==aRow[3]) i4--;  }  else if(sum < target) i3++;  else i4--;  }  }  }  return res;  }  C++2  vector<vector<int>> fourSum(vector<int>& nums, int target) {  vector<vector<int>> res;  vector<int> arow(4);  sort(nums.begin(),nums.end());  int sum, third, fouth, n = nums.size();  for(int first=0; first<n-3; ++first)  {  while(first>0 && nums[first]==arow[0]) ++first;  arow[0] = nums[first];  if(nums[first]+nums[first+1]+nums[first+2]+nums[first+3]>target) break;  if(nums[first]+nums[n-3]+nums[n-2]+nums[n-1]<target) continue;  for(int second=first+1; second<n-2; ++second)  {  while(second>first+1 && nums[second]==arow[1]) ++second;  arow[1] = nums[second];  if(nums[first]+nums[second]+nums[second+1]+nums[second+2]>target) break;  if(nums[first]+nums[second]+nums[n-2]+nums[n-1]<target) continue;  third = second+1;  fouth = n-1;  while(third<fouth)  {  sum = nums[first]+nums[second]+ nums[third]+nums[fouth];  arow[2] = nums[third];  arow[3] = nums[fouth];  if(sum==target)  {  res.push\_back(arow);  while(third<fouth && nums[++third]==arow[2]);  while(third<fouth && nums[--fouth]==arow[3]);  }  else if(sum<target) ++third;  else --fouth;  }  }  }  return res;  } |
| 454. 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 ≤ N ≤ 500. All integers are in the range of -228 to 228 - 1 and the result is guaranteed to be at most 231 - 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  C++1 (399ms)  int fourSumCount(vector<int>& A, vector<int>& B, vector<int>& C, vector<int>& D) {  unordered\_map<int, int> sumAB;  int res = 0;  for(auto a : A)  for(auto b : B)  sumAB[a+b]++;  for(auto c : C)  for(auto d : D)  res += sumAB[-(c+d)];  return res;  }  Testcases:   |  |  |  | | --- | --- | --- | | Id | Testcase | Result | | 1 | [1,2], [-2,-1], [-1,2],[0,2] | 2 | | 2 | [1,2,3], [-2,-1,-2], [-1,2,4],[0,2,-5] | 8 | |
|  |
| 347. 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 ≤ *k* ≤ number of unique elements. * Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size.   C++1 (19ms)  vector<int> topKFrequent(vector<int>& nums, int k) {  unordered\_map<int, int> count;  for(auto num : nums)  count[num]++;  priority\_queue<int, vector<int>, greater<int>> pq;  for(auto ele : count)  {  pq.push(ele.second);  if(pq.size()>k) pq.pop();  }  vector<int>res;  for(auto ele : count)  if(ele.second >= pq.top())  res.push\_back(ele.first);  return res;  }  C++2  vector<int> topKFrequent(vector<int>& nums, int k) {  unordered\_map<int, int> counts;  for(int n : nums) ++counts[n];  vector<vector<int>> buckets(nums.size()+1); // no 0 frequent elements  for(auto m : counts)  {  buckets[m.second].push\_back(m.first);  }  vector<int> res;  for(int i=buckets.size()-1; i>=0; --i)  {  for(int j=0; j<buckets[i].size(); ++j)  {  res.push\_back(buckets[i][j]);  if(res.size()>=k) return res;  }  }  return res;  }  Java1  public List<Integer> topKFrequent(int[] nums, int k) {  Map<Integer, Integer> freqMap = new HashMap<>();  PriorityQueue<Map.Entry<Integer, Integer>> pq =  new PriorityQueue<>((a, b) -> a.getValue() - b.getValue());  List<Integer> res = new ArrayList<>();    for(int num : nums)  {  int count = freqMap.getOrDefault(num, 0);  freqMap.put(num, count + 1);  }    for(Map.Entry<Integer, Integer> map : freqMap.entrySet())  {  pq.offer(map);  if(pq.size()>k)  pq.poll();  }    while(!pq.isEmpty())  {  int num = pq.poll().getKey();  res.add(num);  }  return res;  }  Java2  public List<Integer> topKFrequent(int[] nums, int k) {  int len = nums.length;  Map<Integer, Integer> freqMap = new HashMap<>();  List<Integer> [] buckets = new List[len+1];  List<Integer> res = new ArrayList<>();    for(int num : nums)  {  freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);  }    for(int key : freqMap.keySet())  {  int freq = freqMap.get(key);  if(buckets[freq] == null)  {  buckets[freq] = new ArrayList();  }  buckets[freq].add(key);  }  /\*  for(Map.Entry<Integer, Integer> map : freqMap.entrySet())  {  int freq = map.getValue();  if(buckets[freq] == null)  {  buckets[freq] = new ArrayList();  }  buckets[freq].add(map.getKey());  }  \*/    for(int i=len; i>0; --i)  {  if(buckets[i]==null) continue;  for(int j=0; j<buckets[i].size(); ++j)  {  res.add(buckets[i].get(j));  if(res.size()>= k)  return res;  }  }  return res;  } |
| 532. 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](https://en.wikipedia.org/wiki/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].   C++1 (39ms)  int findPairs(vector<int>& nums, int k) {  if(k<0) return 0;  unordered\_map<int, int> map;  int res = 0;  for(auto num : nums)  map[num]++;  for(auto num : nums)  {  if(k==0)  {  if(map[num]>1) res++;  }  else if(map[num] > 0)  {  if(map[num-k]>0) res++;  if(map[num+k]>0) res++;  }  map[num]=0;  }  return res;  } |
| 525. 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.  C++1 (189ms)  int findMaxLength(vector<int>& nums) {  for(int i=0; i<nums.size(); ++i)  if(nums[i]==0) nums[i]=-1;  unordered\_map<int, int> sumToIndex;  sumToIndex[0]=-1;  int maxL=0, sum=0;  for(int i=0; i<nums.size(); ++i)  {  sum += nums[i];  if(sumToIndex.count(sum)>0)  maxL = max(maxL, i-sumToIndex[sum]);  else  sumToIndex[sum] = i;  }  return maxL;  } |
| 355. 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);  C++1  class Twitter {  private:  unordered\_map<int, unordered\_set<int>> followeree; // follower, followees  unordered\_map<int, map<int, int>> tweetters; // userId, timestamp, tweetId  long long tweetNum;  public:  /\*\* Initialize your data structure here. \*/  Twitter() {  tweetNum = 0;  }    /\*\* Compose a new tweet. \*/  void postTweet(int userId, int tweetId) {  tweetters[userId][tweetNum++] = tweetId;  follow(userId, userId); // if user post a tweet, follow himself  }    /\*\* 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. \*/  vector<int> getNewsFeed(int userId) {  vector<int> res;  map<int, int> top10;  for(auto followee : followeree[userId])  {  for(auto tweet=tweetters[followee].rbegin(); tweet!=tweetters[followee].rend(); ++tweet)  {  if(top10.size()==10 && top10.begin()->first > tweet->first) break;  top10[tweet->first] = tweet->second;  if(top10.size()>10) top10.erase(top10.begin());  }  }  for(auto top=top10.rbegin(); top!=top10.rend(); ++top)  res.push\_back(top->second);  return res;  }    /\*\* Follower follows a followee. If the operation is invalid, it should be a no-op. \*/  void follow(int followerId, int followeeId) {  followeree[followerId].insert(followeeId);  }    /\*\* Follower unfollows a followee. If the operation is invalid, it should be a no-op. \*/  void unfollow(int followerId, int followeeId) {  if(followerId != followeeId)  followeree[followerId].erase(followeeId);  }  };  /\*\*  \* Your Twitter object will be instantiated and called as such:  \* Twitter obj = new Twitter();  \* obj.postTweet(userId,tweetId);  \* vector<int> param\_2 = obj.getNewsFeed(userId);  \* obj.follow(followerId,followeeId);  \* obj.unfollow(followerId,followeeId);  \*/ |
| 432. 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.  C++1  class AllOne {  public:  unordered\_map<string, int> keyToVal;  map<int, unordered\_set<string>> valToKey;  /\*\* Initialize your data structure here. \*/  AllOne() {  keyToVal.clear();  valToKey.clear();  }  void removeValKey(int val, string key)  {  valToKey[val].erase(key);  if(valToKey[val].empty()) valToKey.erase(val);  }  /\*\* Inserts a new key <Key> with value 1. Or increments an existing key by 1. \*/  void inc(string key) {  keyToVal[key]++;  valToKey[keyToVal[key]].insert(key);  if(keyToVal[key]==1) return;  removeValKey(keyToVal[key]-1, key);  }    /\*\* Decrements an existing key by 1. If Key's value is 1, remove it from the data structure. \*/  void dec(string key) {  if(keyToVal.count(key)==0) return;  removeValKey(keyToVal[key], key);  keyToVal[key]--;    if(keyToVal[key]==0)  {  keyToVal.erase(key);  return;  }  valToKey[keyToVal[key]].insert(key);  }    /\*\* Returns one of the keys with maximal value. \*/  string getMaxKey() {  if(valToKey.empty()) return "";  return \*(valToKey.rbegin()->second.begin());  }    /\*\* Returns one of the keys with Minimal value. \*/  string getMinKey() {  if(valToKey.empty()) return "";  return \*(valToKey.begin()->second.begin());  }  };  /\*\*  \* Your AllOne object will be instantiated and called as such:  \* AllOne obj = new AllOne();  \* obj.inc(key);  \* obj.dec(key);  \* string param\_3 = obj.getMaxKey();  \* string param\_4 = obj.getMinKey();  \*/ |
| 594. Longest Harmonious Subsequence  We define a harmonious array is an array where the difference between its maximum value and its minimum value is **exactly** 1.  Now, given an integer array, you need to find the length of its longest harmonious subsequence among all its possible [subsequences](https://en.wikipedia.org/wiki/Subsequence).  **Example 1:**  **Input:** [1,3,2,2,5,2,3,7]  **Output:** 5  **Explanation:** The longest harmonious subsequence is [3,2,2,2,3].  **Note:** The length of the input array will not exceed 20,000.  C++1  int findLHS(vector<int>& nums) {  map<int, int> count;  int res=0;  for(int num : nums)  count[num]++;  for(auto c : count)  if(count.count(c.first + 1) > 0)  res = max(res, c.second + count[c.first + 1]);  return res;  }  C++2  int findLHS(vector<int>& nums) {  map<int, int> count;  int res=0;  for(int num : nums)  count[num]++;  for(auto c = count.begin(); c!=count.end(); ++c)  if(count.count(c->first + 1) > 0)  res = max(res, c->second + count[c->first + 1]);  return res;  } |
| 166. 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)".   C++1  string fractionToDecimal(int numerator, int denominator) {  if(numerator==0) return "0";  long n = numerator, d = denominator;  string res;  if(n<0 ^ d<0) res += "-";  n = abs(n);  d = abs(d);  res += to\_string(n/d);  if(n%d == 0) return res;  res += ".";  unordered\_map<int, int> map;  for(long i=n%d; i!=0; i%=d)  {  if(map.count(i) > 0)  {  res.insert(map[i],1, '(');  res += ")";  break;  }  map[i] = res.size();  i \*= 10;  res += to\_string(i/d);  }  return res;  } |
| 599. Minimum Index Sum of Two Lists  Suppose Andy and Doris want to choose a restaurant for dinner, and they both have a list of favorite restaurants represented by strings.  You need to help them find out their **common interest** with the **least list index sum**. If there is a choice tie between answers, output all of them with no order requirement. You could assume there always exists an answer.  **Example 1:**  **Input:**  ["Shogun", "Tapioca Express", "Burger King", "KFC"]  ["Piatti", "The Grill at Torrey Pines", "Hungry Hunter Steakhouse", "Shogun"]  **Output:** ["Shogun"]  **Explanation:** The only restaurant they both like is "Shogun".  **Example 2:**  **Input:**  ["Shogun", "Tapioca Express", "Burger King", "KFC"]  ["KFC", "Shogun", "Burger King"]  **Output:** ["Shogun"]  **Explanation:** The restaurant they both like and have the least index sum is "Shogun" with index sum 1 (0+1).  **Note:**   1. The length of both lists will be in the range of [1, 1000]. 2. The length of strings in both lists will be in the range of [1, 30]. 3. The index is starting from 0 to the list length minus 1. 4. No duplicates in both lists.   C++1  vector<string> findRestaurant(vector<string>& list1, vector<string>& list2) {  unordered\_map<string, int> map1, map2, commonMap;  for(int i=0; i<list1.size(); ++i)  map1[list1[i]] = i;  for(int j=0; j<list2.size(); ++j)  map2[list2[j]] = j;    int indexSum = 2000;  for(string r : list1)  {  if(map2.count(r) > 0)  {  commonMap[r] = map1[r] + map2[r];  indexSum = min(indexSum, commonMap[r]);  }  }    vector<string> res;  for(auto id : commonMap)  if(id.second == indexSum)  res.push\_back(id.first);  return res;  } |
|  |

# **解题思路**

## 举例法

|  |
| --- |
| 343. 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 and not larger than 58.  **Hint:**   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.   C++1 (0ms)  int integerBreak(int n) {  if(n==2) return 1;  if(n==3) return 2;  int mo = n%3;  if(mo==0) return pow(3, n/3);  if(mo==1) return pow(3, n/3 - 1) \* 4;  if(mo==2) return pow(3, n/3) \* 2;  }  C++2 (0ms)  int integerBreak(int n) {  if(n<=3) return n-1;  if(n==4) return 4;  int group = n/3, mod = n%3, res = 1;  if(mod == 1)  {  res = 4;  group--;  }  if(mod == 2) res = 2;  for(int i=0; i<group; ++i) res\*=3;  return res;  }  Java1  public int integerBreak(int n) {  if(n<=3) return n-1;  if(n==4) return 4;  int group = n/3;  int mod = n%3;  int res = 1;  if(mod == 1)  {  res = 4;  group--;  }  else if(mod == 2)  res = 2;  for(int i=0; i<group; ++i)  res \*= 3;  return res;  } |
|  |
|  |
|  |

## 模式匹配法

## 简化推广法

## 简单构造法

## 数据结构头脑风暴法

## Iterator

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 341. 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].  C++1 (19ms)  /\*\*  \* // This is the interface that allows for creating nested lists.  \* // You should not implement it, or speculate about its implementation  \* class NestedInteger {  \* public:  \* // Return true if this NestedInteger holds a single integer, rather than a nested list.  \* bool isInteger() const;  \*  \* // Return the single integer that this NestedInteger holds, if it holds a single integer  \* // The result is undefined if this NestedInteger holds a nested list  \* int getInteger() const;  \*  \* // Return the nested list that this NestedInteger holds, if it holds a nested list  \* // The result is undefined if this NestedInteger holds a single integer  \* const vector<NestedInteger> &getList() const;  \* };  \*/  class NestedIterator {  public:  stack<NestedInteger> myList;  void PushListIntoStack(vector<NestedInteger> &nestedList)  {  for(int i=nestedList.size()-1; i>=0; --i)  myList.push(nestedList[i]);  }  NestedIterator(vector<NestedInteger> &nestedList) {  PushListIntoStack(nestedList);  }  int next() {  NestedInteger top = myList.top();  myList.pop();  return top.getInteger();  }  bool hasNext() {  if(myList.empty()) return false;  NestedInteger top = myList.top();  if(top.isInteger()) return true;  myList.pop();  PushListIntoStack(top.getList());  return hasNext();  }  };  /\*\*  \* Your NestedIterator object will be instantiated and called as such:  \* NestedIterator i(nestedList);  \* while (i.hasNext()) cout << i.next();  \*/  Java1 (10ms)  public class NestedIterator implements Iterator<Integer> {    private Stack<NestedInteger> stk;    private void pushListToStack(List<NestedInteger> nestedList){  for(int i=nestedList.size()-1; i>=0; --i)  stk.push(nestedList.get(i));  }  public NestedIterator(List<NestedInteger> nestedList) {  stk = new Stack<NestedInteger>();  pushListToStack(nestedList);  }  @Override  public Integer next() {  return stk.pop().getInteger();  }  @Override  public boolean hasNext() {  while(!stk.empty())  {  NestedInteger top = stk.peek();  if(top.isInteger())  return true;  stk.pop();  List<NestedInteger> openList = top.getList();  pushListToStack(openList);  }  return false;  }  }  Testcase:   |  |  |  | | --- | --- | --- | | Id | Testcase | Result | | 1 | [[1,1],[6,7],[2,3]] | [1,1,6,7,2,3] | | 2 | [[-1,-1],[[[]]],[2,3]] | [-1,-1,2,3] | |
| 173. 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.  /\*\*  \* Your BSTIterator will be called like this:  \* BSTIterator i = BSTIterator(root);  \* while (i.hasNext()) cout << i.next();  \*/  C++1 (19ms) using stack  class BSTIterator {  public:  stack<TreeNode\*> left;  void PushLeftToStack(TreeNode\* root)  {  while(root!=NULL)  {  left.push(root);  root=root->left;  }  }  BSTIterator(TreeNode \*root) {  PushLeftToStack(root);  }  /\*\* @return whether we have a next smallest number \*/  bool hasNext() {  return !left.empty();  }  /\*\* @return the next smallest number \*/  int next() {  TreeNode\* cur = left.top();  left.pop();  int res = cur->val;    cur = cur->right;  PushLeftToStack(cur);  return res;  }  };  C++2  class BSTIterator {  private:  stack<TreeNode\*> myStack;  void pushLeft(TreeNode\* root)  {  for(;root!=NULL; myStack.push(root), root = root->left);  }  public:  BSTIterator(TreeNode \*root) {  pushLeft(root);  }  /\*\* @return whether we have a next smallest number \*/  bool hasNext() {  return !myStack.empty();  }  /\*\* @return the next smallest number \*/  int next() {  TreeNode\* t = myStack.top();  myStack.pop();  pushLeft(t->right);  return t->val;  }  };  Java1  public class BSTIterator {  private Stack<TreeNode> smallList;  public BSTIterator(TreeNode root) {  smallList = new Stack<TreeNode>();  storeLeft(root);  }    private void storeLeft(TreeNode root)  {  while(root != null)  {  smallList.push(root);  root = root.left;  }  }  /\*\* @return whether we have a next smallest number \*/  public boolean hasNext() {  return !smallList.empty();  }  /\*\* @return the next smallest number \*/  public int next() {  TreeNode cur = smallList.pop();  storeLeft(cur.right);  return cur.val;  }  } |
| 284. 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.  **Follow up**: How would you extend your design to be generic and work with all types, not just integer?  C++1  // Below is the interface for Iterator, which is already defined for you.  // \*\*DO NOT\*\* modify the interface for Iterator.  class Iterator {  struct Data;  Data\* data;  public:  Iterator(const vector<int>& nums);  Iterator(const Iterator& iter);  virtual ~Iterator();  // Returns the next element in the iteration.  int next();  // Returns true if the iteration has more elements.  bool hasNext() const;  };  class PeekingIterator : public Iterator {  private:  int curNext;  bool curHasNext;  public:  void getCurValues()  {  curHasNext = Iterator::hasNext();  if(curHasNext) curNext = Iterator::next();  }    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.  getCurValues();  }    // Returns the next element in the iteration without advancing the iterator.  int peek() {  return curNext;  }  // hasNext() and next() should behave the same as in the Iterator interface.  // Override them if needed.  int next() {  int cur = curNext;  getCurValues();  return cur;  }  bool hasNext() const {  return curHasNext;  }  }; |
| 604. Design Compressed String Iterator  Design and implement a data structure for a compressed string iterator. It should support the following operations: next and hasNext.  The given compressed string will be in the form of each letter followed by a positive integer representing the number of this letter existing in the original uncompressed string.  next() - if the original string still has uncompressed characters, return the next letter; Otherwise return a white space. hasNext() - Judge whether there is any letter needs to be uncompressed.  **Note:** Please remember to **RESET** your class variables declared in StringIterator, as static/class variables are **persisted across multiple test cases**. Please see [here](https://leetcode.com/faq/#different-output) for more details.  **Example:**  StringIterator iterator = new StringIterator("L1e2t1C1o1d1e1");  iterator.next(); // return 'L'  iterator.next(); // return 'e'  iterator.next(); // return 'e'  iterator.next(); // return 't'  iterator.next(); // return 'C'  iterator.next(); // return 'o'  iterator.next(); // return 'd'  iterator.hasNext(); // return true  iterator.next(); // return 'e'  iterator.hasNext(); // return false  iterator.next(); // return ' '  C++1  class StringIterator {    private:  vector<int> str;  int cur;  int len;    public:  vector<int> Parse(string cs)  {  vector<int> res;  int count=0;  for(int i=0; i<cs.length(); ++i)  {  if(isdigit(cs[i]))  {  count = 10\*count + (cs[i]-'0');  if(i+1==cs.length() || !isdigit(cs[i+1]))  {  res.push\_back(count);  count=0;  }  }  else  res.push\_back(cs[i]);  }  return res;  }    StringIterator(string compressedString) {  str = Parse(compressedString);  cur = 0;  len = str.size();  }    char next() {  if(hasNext()){  char res = str[cur];  if(--str[cur+1] == 0)  cur += 2;  return res;  }  return ' ';  }    bool hasNext() {  return cur < len;  }  };  /\*\*  \* Your StringIterator object will be instantiated and called as such:  \* StringIterator obj = new StringIterator(compressedString);  \* char param\_1 = obj.next();  \* bool param\_2 = obj.hasNext();  \*/ |
|  |

## Th

|  |
| --- |
| 400. Nth Digit  Find the *n*th 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* < 231).  **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.  C++1 (3ms)  int findNthDigit(int n) {  int len=1, start =1;  long count = 9;  while(n > len\*count)  {  n -= len\*count; //minus the length of a kind of number (1digit, 2 digits,….)  len++; // (1digit, 2 digits,….)  count \*= 10; //the count of the same digit number (9, 90 ,900, …)  start \*= 10; //the start number of the same digit number (1, 10, 100…)  }  start += (n-1)/len;  string res = to\_string(start);  return res[(n-1)%len] - '0';  } |
| 414. Third Maximum Number  Given a **non-empty** array of integers, return the **third** maximum number in this array. If it does not exist, return the maximum number. The time complexity must be in O(n).  **Example 1:**  **Input:** [3, 2, 1]  **Output:** 1  **Explanation:** The third maximum is 1.  **Example 2:**  **Input:** [1, 2]  **Output:** 2  **Explanation:** The third maximum does not exist, so the maximum (2) is returned instead.  **Example 3:**  **Input:** [2, 2, 3, 1]  **Output:** 1  **Explanation:** Note that the third maximum here means the third maximum distinct number.  Both numbers with value 2 are both considered as second maximum.  C++1 (6ms)  int thirdMax(vector<int>& nums) {  long first=LONG\_MIN, second=LONG\_MIN, third=LONG\_MIN;  for(auto num : nums)  {  if(num>first)  {  third = second;  second = first;  first=num;  }  else if(num < first && num > second)  {  third = second;  second = num;  }  else if (num < second && num > third)  third = num;  }  return (third==LONG\_MIN) ? first : third;  } |
| 215. Kth Largest Element in an Array  Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.  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.  C++1 (9ms)  int findKthLargest(vector<int>& nums, int k) {  priority\_queue<int, vector<int>, greater<int>> q;  for(auto num : nums)  {  q.push(num);  if(q.size()>k) q.pop();  }  return q.empty() ? -1 : q.top();  }  C++2 (20ms)  int findKthLargest(vector<int>& nums, int k) {  priority\_queue<int> maxq(nums.begin(),nums.end());  for(int i=0; i<k-1; ++i) maxq.pop();  return maxq.top();  }  C++3 (28ms)  int findKthLargest(vector<int>& nums, int k) {  multiset<int> mset;  for(int n : nums)  {  mset.insert(n);  if(mset.size()>k) mset.erase(mset.begin());  }  return \*mset.begin();  }  Java1 (13ms)  public int findKthLargest(int[] nums, int k) {  PriorityQueue<Integer> q = new PriorityQueue<>();  for(int i=0; i<nums.length; ++i)  {  q.offer(nums[i]);  if(i < k) continue;  q.poll();  }  return q.peek();  } |
| 313. 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 ≤ 106, 0 < primes[i] < 1000. (4) The nth super ugly number is guaranteed to fit in a 32-bit signed integer.  C++1  int nthSuperUglyNumber(int n, vector<int>& primes) {  int pn = primes.size();  vector<int> uglys(n);  vector<int> uglyId(pn);  vector<int> pVal(pn, 1);  int next = 1;  for(int i=0; i<n; ++i)  {  uglys[i] = next;  next = INT\_MAX;  for(int j=0; j<pn; ++j)  {  if(uglys[i] == pVal[j])  pVal[j] = uglys[uglyId[j]++] \* primes[j]; //multiply uglys from beginning  next = min(next, pVal[j]); //update the min val  }  }  return uglys[n-1];  } |
| 264. 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, and *n* **does not exceed 1690**.  C++1 (TLE)  class Solution {  public:  vector<int> factors = {2,3,5};  bool isUgly(int num)  {  for(int i=0; i<3; ++i)  {  if(num % factors[i] == 0)  {  num /= factors[i];  i--;  }  }  return num==1;  }  int nthUglyNumber(int n) {  static vector<int> uglyNums(1,1);  for(int i=uglyNums.back()+1; uglyNums.size()<n ; ++i)  {  if(isUgly(i))  {  uglyNums.push\_back(i);  }  }  return uglyNums[n-1];  }  };  C++2  int nthUglyNumber(int n) {  vector<int> res(1,1);  int p2=0, p3=0, p5=0;  while(res.size()<n)  {  res.push\_back(min(res[p2]\*2, min(res[p3]\*3, res[p5]\*5)));  if(res.back()==res[p2]\*2) ++p2;  if(res.back()==res[p3]\*3) ++p3;  if(res.back()==res[p5]\*5) ++p5;  }  return res[n-1];  }  C++3  int nthUglyNumber(int n) {  static vector<int> uglyNums(1,1);  static int two(0), three(0), five(0);  while(uglyNums.size()<n)  {  int nextMin = min(uglyNums[two]\*2, min(uglyNums[three]\*3, uglyNums[five]\*5));  uglyNums.push\_back(nextMin);  if(uglyNums[two]\*2 == nextMin) two++;  if(uglyNums[three]\*3 == nextMin) three++;  if(uglyNums[five]\*5 == nextMin) five++;  }  return uglyNums[n-1];  } |
|  |

## Visited

|  |
| --- |
| 417. 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 parentheses in above matrix).  C++1  class Solution {  public:  void dfsOceans(vector<vector<int>>& matrix, vector<vector<bool>>& visited, int i, int j, int height)  {  if(i<0 || i>=matrix.size() || j<0 || j>=matrix[0].size() || visited[i][j]) return;  if(matrix[i][j] < height) return;  visited[i][j] = true;  dfsOceans(matrix, visited, i-1, j, matrix[i][j]);  dfsOceans(matrix, visited, i+1, j, matrix[i][j]);  dfsOceans(matrix, visited, i, j-1, matrix[i][j]);  dfsOceans(matrix, visited, i, j+1, matrix[i][j]);  }  vector<pair<int, int>> pacificAtlantic(vector<vector<int>>& matrix) {  vector<pair<int, int>> res;  int h = matrix.size();  int w = (h>0) ? matrix[0].size() : 0;  vector<vector<bool>> Pacific(h, vector<bool>(w));  vector<vector<bool>> Atlantic(h, vector<bool>(w));  for(int i=0; i<h; ++i)  {  dfsOceans(matrix, Pacific, i, 0, matrix[i][0]);  dfsOceans(matrix, Atlantic, i, w-1, matrix[i][w-1]);  }  for(int j=0; j<w; ++j)  {  dfsOceans(matrix, Pacific, 0, j, matrix[0][j]);  dfsOceans(matrix, Atlantic, h-1, j, matrix[h-1][j]);  }  for(int i=0; i<h; ++i)  for(int j=0; j<w; ++j)  if(Pacific[i][j] && Atlantic[i][j])  res.push\_back(make\_pair(i, j));  return res;  }  }; |
| 568. Maximum Vacation Days  LeetCode wants to give one of its best employees the option to travel among **N** cities to collect algorithm problems. But all work and no play makes Jack a dull boy, you could take vacations in some particular cities and weeks. Your job is to schedule the traveling to maximize the number of vacation days you could take, but there are certain rules and restrictions you need to follow.  **Rules and restrictions:**   1. You can only travel among **N** cities, represented by indexes from 0 to N-1. Initially, you are in the city indexed 0 on **Monday**. 2. The cities are connected by flights. The flights are represented as a **N\*N** matrix (not necessary symmetrical), called **flights** representing the airline status from the city i to the city j. If there is no flight from the city i to the city j, **flights[i][j] = 0**; Otherwise, **flights[i][j] = 1**. Also, **flights[i][i] = 0** for all i. 3. You totally have **K** weeks (**each week has 7 days**) to travel. You can only take flights at most once **per day** and can only take flights on each week's **Monday** morning. Since flight time is so short, we don't consider the impact of flight time. 4. For each city, you can only have restricted vacation days in different weeks, given an **N\*K** matrix called **days** representing this relationship. For the value of **days[i][j]**, it represents the maximum days you could take vacation in the city **i** in the week **j**.   You're given the **flights** matrix and **days** matrix, and you need to output the maximum vacation days you could take during **K** weeks.  **Example 1:**  **Input:**flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[1,3,1],[6,0,3],[3,3,3]]  **Output:** 12  **Explanation:**  Ans = 6 + 3 + 3 = 12.  One of the best strategies is:  1st week : fly from city 0 to city 1 on Monday, and play 6 days and work 1 day.  (Although you start at city 0, we could also fly to and start at other cities since it is Monday.)  2nd week : fly from city 1 to city 2 on Monday, and play 3 days and work 4 days.  3rd week : stay at city 2, and play 3 days and work 4 days.  **Example 2:**  **Input:**flights = [[0,0,0],[0,0,0],[0,0,0]], days = [[1,1,1],[7,7,7],[7,7,7]]  **Output:** 3  **Explanation:**  Ans = 1 + 1 + 1 = 3.  Since there is no flights enable you to move to another city, you have to stay at city 0 for the whole 3 weeks.  For each week, you only have one day to play and six days to work.  So the maximum number of vacation days is 3.  **Example 3:**  **Input:**flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[7,0,0],[0,7,0],[0,0,7]]  **Output:** 21  **Explanation:** Ans = 7 + 7 + 7 = 21  One of the best strategies is:  1st week : stay at city 0, and play 7 days.  2nd week : fly from city 0 to city 1 on Monday, and play 7 days.  3rd week : fly from city 1 to city 2 on Monday, and play 7 days.  **Note:**   1. **N and K** are positive integers, which are in the range of [1, 100]. 2. In the matrix **days**, all the values are integers in the range of [0, 1]. 3. In the matrix **flights**, all the values are integers in the range [0, 7]. 4. You could stay at a city beyond the number of vacation days, but you should **work** on the extra days, which won't be counted as vacation days. 5. If you fly from the city A to the city B and take the vacation on that day, the deduction towards vacation days will count towards the vacation days of city B in that week. 6. We don't consider the impact of flight hours towards the calculation of vacation days.   C++1 (ETL)  class Solution {  public:  int vacation=0, city, week;  void dfs(vector<vector<int>>& flights, vector<vector<int>>& days, vector<bool> visited, int c, int w, int sum)  {  if(w==week)  {  vacation = max(vacation, sum);  return;  }  visited[c] = true;  for(int i=0;i<city; ++i)  {  if(i==c || (flights[c][i]==1 && !visited[i]))  dfs(flights, days, visited, i, w+1, sum+days[i][w]);  }  }  int maxVacationDays(vector<vector<int>>& flights, vector<vector<int>>& days) {  city = days.size(), week = days[0].size();  vector<bool> visited(city);  dfs(flights, days, visited, 0, 0, 0);  return vacation;  }  }; |
|  |

# END

## Unsolved

|  |
| --- |
| Codility [ArrayInversionCount](https://codility.com/demo/results/trainingU2HM9V-B8Z/)  A zero-indexed array A consisting of N integers is given. An *inversion* is a pair of indexes (P, Q) such that P < Q and A[Q] < A[P].  Write a function:  int solution(vector<int> &A);  that computes the number of inversions in A, or returns −1 if it exceeds 1,000,000,000.  Assume that:   * N is an integer within the range [0..100,000]; * each element of array A is an integer within the range [−2,147,483,648..2,147,483,647].   For example, in the following array:  A[0] = -1 A[1] = 6 A[2] = 3 A[3] = 4 A[4] = 7 A[5] = 4  there are four inversions:  (1,2) (1,3) (1,5) (4,5)  so the function should return 4.  Complexity:   * expected worst-case time complexity is O(N\*log(N)); * expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments).   Elements of input arrays can be modified.  C++1 (63%)  **int** **solution**(**vector**<**int**> &A) {  // write your code in C++14 (g++ 6.2.0)  **if**(A.empty()) **return** 0;  **int** res = 0;  **multiset**<**int**> **set**(A.begin(), A.end());  **for**(**unsigned** **int** i=A.size()-1; i>0; --i)  {  **auto** it = **set**.upper\_bound(A[i]);  res += distance(it, **set**.end());  **if**(res>1e9) **return** -1;  **set**.erase(--it);  }  **return** res;  }  wrong  public int solution(int[] A) {  // write your code in Java SE 8  List<Integer> count = new LinkedList<>();  int res = 0;  for(int a : A)  count.add(a);  Collections.sort(count);  for(int a : A){  count.remove(new Integer(a));  int id = Collections.binarySearch(count, a);  int smallerNum = id>0 ? id-1 : id==0 ? 0 : Math.abs(id+1);  res += smallerNum;  if(res<0 || res > 1\_000\_000\_000) return -1;  }  return res;  } |
|  |

