# Print Triplets which sums to Zero.

```
void triplets(vector<int>& arr) {
    int sum = 0;
    unordered_map<int, int> map;
    for(int i=0;i<arr.size();i++) {
        for(int j=i+1;j<arr.size();j++) {
            int tsum = arr[i] + arr[j];
            int rsum = sum - tsum;
            if(map.find(rsum) == map.end()) {
                auto k = map[rsum];
                if(i != j && i != k) {
                    cout<<i<<" "<<j<<" "<<k<<endl;
                }
            } else {
                map[arr[i]] = i;
                map[arr[j]] = j;
            }
        }
    }
}
```

```
void triplets(vector<int>& arr) {
    int sum = 0;
    sort(arr.begin(), arr.end());

    for(int i=0;i<arr.size();i++) {
        int j = i+1;
        int k = arr.size()-1;

        while(j < k) {
            int curSum = arr[i] + arr[j] + arr[k];
            if(curSum == sum) {
                cout<<arr[i]<<" "<<arr[j]<<" "<<arr[k]<<endl;
                j++;k--;
            } else if(curSum < sum) {
                j++;
            } else {
                k--;
            }
        }
    }
}
```

## Given n, output the numbers from 0 to 2^n-1 (inclusive) in n-bit binary form, in such an order that adjacent numbers in the list differ by exactly 1 bit.

Just do a loop from 0 to 2^n-1 with the following.

```
unsigned int binaryToGray(unsigned int num)
{
    return (num >> 1) ^ num;
}
```

## Subarray Product Less Than K

☑ Solved

```
int numSubarrayProductLessThanK(vector<int>& nums, int k) {
    int sum = 1;
    int count = 0;

    for(int i=0,j=0;j<nums.size();j++) {
        sum *= nums[j];
        while(i <= j && sum >= k) {
            sum /= nums[i++];
        }

        count += (j-i+1);
    }

    return count;
}
```

## Two Sum

☑ Solved

```
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> map;
    int index = 0;
    for(auto& num : nums) {
        int rsum = target - num;
        if(map.find(rsum) != map.end()) {
            return {map[rsum], index};
        } else {
            map[num] = index;
        }

        index++;
    }

    return {-1,-1};
}
```

# Longest Substring Without Repeating Characters

☑ Solved

```
int lengthOfLongestSubstring(string s) {
    vector<int> map(256, -1);
    int i = 0, j =0;
    int mxLen = 0;

    for(;j<s.length();j++) {
        char ch = s[j];
        if(map[ch] != -1) { // if this character is already seen
            mxLen = max(mxLen, j-i); // get current max len
            while(i<=map[ch]) { // remove the characters <= map[ch]
                char ch1 = s[i];
                map[ch1] = -1;
                i++;
            }
        }

        map[ch] = j; // mark current character
    }
    mxLen = max(mxLen, j-i); // get max length from end

    return mxLen;
}
```

# Square root of a number using binary search

☑ Solved

### *Number square root*

```
int squareRoot(int num) {
    int l = 1, r = num;
    int res = -1;

    while(l <= r) { // do binary search
        int m = l + (r-l)/2; // to avoid overflow
        if(m <= num/m) { // to avoid overflow
            res = m; // always consider the smaller integer before it
            l = m+1; // keep looking for more accurate result
        } else {
            r = m-1; // ignore the values which result in > num and look on left.
        }
    }

    return res;
}
```

### *Floating point square root*

```
float squareRoot(float n) {
    double x = 1, y = n;
    double e = 0.00001;

    // keep doing binary search between 2 numbers until the differnce
    // becomes less than expected diff.
    // first number = (first number + second number) / 2
    // second number = number / first number
    while(y-x > e) {
        x = (x+y)/2.0;
        y = n/x;
    }

    return x;
}
```

# Find leaves from preorder of BST.

```
vector<int> getLeaves(vector<int> &pre) {
    vector<int> leaves;
    stack<int> st;

    int prev = 0;
    for (auto &e : pre) {
        // find the ancestor of current node which is greater than current node.
        int ancestor = 0;
        while (!st.empty() && st.top() < e) {
            ancestor = st.top();
            st.pop();
        }

        // if ancestor is greater than prev node, add it to leaves.
        if (ancestor > prev) {
            leaves.push_back(prev);
        }

        // push the current node to stack and set it to prev.
        st.push(e);
        prev = e;
    }

    // insert the last node.
    if (!st.empty()) {
        leaves.push_back(prev);
    }

    return leaves;
}
```

# Construct BST from preorder

```
Node *constructBst(vector<int>& pre)
{
    int n = pre.size();
    return constructBst(pre, 0, pre[0], INT_MIN, INT_MAX, n);
}
Node *constructBst(vector<int>& pre, int preIndex, int key, int mn, int mx, int sz) {
    if(preIndex >= sz) return nullptr;

    Node *root = nullptr;
    if(key > mn && key < mx) {
        root = new Node(key);
        preIndex++;

        if(preIndex < sz) root->left = constructBst(pre, preIndex, pre[preIndex], mn, key, sz);
        if(preIndex < sz) root->right = constructBst(pre, preIndex, pre[preIndex],key, mx, sz);
    }

    return root;
}
```

# Kth Largest in Array

## Kth largest using sorting

***Complexity: O(nlogn)***

```
int kthLargestUsingSorting(vector<int>& arr, int k) {
    sort(arr.begin(), arr.end(), [](const auto& f, const auto& s) { return f > s; });

    return arr[k - 1];
}
```

## Kth largest using max heap

***Complexity: O(nlogn)***

```
template <class T>
class Heap {
    vector<T> arr;
    int sz = 0;
    std::function<bool(const T&, const T&)> compare;

    public:
    Heap(std::function<bool(const T&, const T&)> compare) : compare(compare) {}

    void push(T e) {
        arr.push_back(e);
        sz++;

        upHeapify(sz - 1);
    }

    T top() {
        return arr[0];
    }

    T pop() {
        auto e = arr[0];
        arr[0] = arr[sz - 1];
        sz--;

        downHeapify(0);

        return e;
    }

    bool empty() {
        return sz == 0;
    }

    private:
    void upHeapify(int idx) {
        T p = parent(idx);
        while (idx != 0 && compare(arr[p], arr[idx])) {
            swap(arr[idx], arr[p]);
            idx = p;
            p = parent(idx);
        }
    }

    void downHeapify(int idx) {
        int largestIndex = idx;

        int l = left(idx);
        if (l < sz && compare(arr[largestIndex], arr[l])) largestIndex = l;
        int r = right(idx);
        if (r < sz && compare(arr[largestIndex], arr[r])) largestIndex = r;

        if (idx != largestIndex) {
            swap(arr[idx], arr[largestIndex]);

            downHeapify(largestIndex);
        }
    }

    int parent(int idx) { return (idx - 1) / 2; }
    int left(int idx) { return 2 * idx + 1; }
    int right(int idx) { return 2 * idx + 2; }
};

int kthLargestUsingHeap(vector<int>& arr, int k) {
    Heap<int> heap([](const auto& f, const auot& s) { return f < s; });
    for (auto& e : arr) heap.push(e);

    int elem = INT_MIN;
    while (!heap.empty() && k--) {
        elem = heap.pop();
    }

    return k > 0 ? INT_MIN : elem;
}
```

## Kth largest using selection algorithm

***Average=O(n), Max = O(n^2)***

```
int kthLargest(vector<int>& arr, int k) {
    return kthLargest(arr, 0, arr.size() - 1, k);
}

int kthLargest(vector<int>& arr, int l, int r, int k) {
    if (l > r) return INT_MIN;

    int p = partition(arr, l, r);
    int q = r - p + 1;  // larger elements on right

    if (k == q)
        return arr[p];
    else if (q < k)
        return kthLargest(arr, l, p - 1, k - q);
    else
        return kthLargest(arr, p + 1, r, k);
}

int partition(vector<int>& arr, int l, int r) {
    int pivot = arr[r];
    int low = l;
    for (int i = l; i < r; i++) {
        if (arr[i] <= pivot) {
            swap(arr[low++], arr[i]);
        }
    }
    swap(arr[low], arr[r]);

    return low;
}
```

# Print k largest elements

```
void printKLargestElements(vector<int>& arr, int k) {
    auto pos = kthLargestIndex(arr, 0, arr.size() - 1, k);
    for (int i = pos; i < arr.size(); i++) cout << arr[i] << " ";
    cout << endl;
}
int kthLargestIndex(vector<int>& arr, int l, int r, int k) {
    if (l > r) return INT_MIN;

    int p = partition(arr, l, r);
    int q = r - p + 1;  // larger elements on right

    if (k == q)
        return p;
    else if (q < k)
        return kthLargestIndex(arr, l, p - 1, k - q);
    else
        return kthLargestIndex(arr, p + 1, r, k);
}
```

# Is Valid parenthesis

```
bool isValid(string s) {
    stack<char> st;
    for(auto ch : s) {
        if(isOpen(ch)) st.push(ch);
        else {
            if(st.empty()) return false;
            if(!isMatch(st.top(), ch)) return false;
            st.pop();
        }
    }

    return st.empty();
}

bool isOpen(char ch) {
    return ch == '[' || ch == '{' || ch == '(';
}

bool isMatch(char open, char close) {
    return (open == '[' && close == ']')
            || (open == '{' && close == '}')
            || (open == '(' && close == ')');
}
```

# Minimum number of meeting rooms required.

### *Approach 1*

```
int minNumberOfMeetingRooms1(vector<Interval>& meetings) {
    priority_queue<int, vector<int>, greater<int>> minHeap;
    sort(meetings.begin(), meetings.end(), [](auto& f, auto& s) { return f.start < s.start; });

    for (auto& meeting : meetings) {
        if (!minHeap.empty() && meeting.start >= minHeap.top()) {
            minHeap.pop();
        }
        minHeap.push(meeting.end);
    }

    return minHeap.size();
}
```

### *Aproach 2*

```
void minNumberOfMeetingRooms2(vector<Interval>& meetings) {
    auto comparator = [](const auto& f, const auto& s) { return f.end < s.end; };
    list<priority_queue<Interval, vector<Interval>, decltype(comparator)>> minHeaps;
    sort(meetings.begin(), meetings.end(), [](auto& f, auto& s) { return f.start < s.start; });

    for (auto& meeting : meetings) {
        bool found = false;
        for (auto& q : minHeaps) {
            if (!q.empty() && meeting.start >= q.top().end) {
                q.push(meeting);
                found = true;
                break;
            }
        }

        if (!found) {
            minHeaps.push_back({});
            minHeaps.back().push(meeting);
        }
    }

    for (auto& q : minHeaps) {
        while (!q.empty()) {
            auto interval = q.top();
            cout << interval.to_string() << " ";
            q.pop();
        }
        cout << endl;
    }

    cout << format("Minimum number of meeting rooms required={}", minHeaps.size()) << endl;
}
```

### ***Approach 3***

```
private:
int minNumberOfMeetingRooms3(vector<Interval>& meetings) {
    vector<pair<int, char>> vals;

    for (auto& meeting : meetings) {
        vals.push_back({meeting.start, 'f'});
        vals.push_back({meeting.end, 's'});
    }

    sort(vals.begin(), vals.end(), [](const auto& f, const auto& s) {
        if (f.first < s.first) return true;
        if (f.first > s.first) return false;
        return f.second < s.second;
    });

    int minRooms = 0;
    int cur = 0;
    for (auto& val : vals) {
        if (val.second == 'f')
            cur++;
        else {
            minRooms = max(minRooms, cur);
            cur--;
        }
    }

    return minRooms;
}
```

# Maximum sum subsequence of non-consecutive numbers.

```
int maxSumSubsequence(vector<int>& nums) {
    int n = nums.size();
    if(n == 0) return 0;
    if(n == 1) return nums[0];

    vector<int> table(n);
    table[0] = nums[0];
    table[1] = max(nums[0], nums[1]);

    for(int i=2;i<n;i++) {
        table[i] = max(table[i-1], nums[i] + table[i-2]);
    }

    return table[n-1];
}
```

# Maximum sum subsequence without consecutive numbers

```
int maxSumSubsequenceRecursive(vector<int>& nums) {
    return maxSumSubsequenceRecursive(nums, 0, nums.size(), 0);
}

int maxSumSubsequenceRecursive(vector<int>& nums, int idx, int sz, int cur) {
    if (idx == sz) return 0;
    if (idx == sz - 1) return nums[idx];
    if (idx == sz - 2) return max(nums[idx], nums[idx + 1]);

    return max(maxSumSubsequenceRecursive(nums, idx + 1, sz, cur),
               nums[idx] + maxSumSubsequenceRecursive(nums, idx + 2, sz, cur));
}
```

```
int maxSumSubsequence(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;
    if (n == 1) return nums[0];

    vector<int> table(n);
    table[0] = nums[0];
    table[1] = max(nums[0], nums[1]);

    for (int i = 2; i < n; i++) {
        table[i] = max(table[i - 1], nums[i] + table[i - 2]);
    }

    return table[n - 1];
}
```

## Print combinations

```
https://careercup.com/question?id=5634222671790080

Given a string as input, return the list of all the patterns possible:


'1' : ['A', 'B', 'C'],
'2' : ['D', 'E'],
'12' : ['X']
'3' : ['P', 'Q']
Example if input is '123', then output should be [ADP, ADQ, AEP, AEQ, BDP, BDQ, BEP, BEQ, CDP, C
```

```
void printCombinations(string str) {
    unordered_map<string, list<string>> map = {
        {"1", {"A", "B", "C"}},
        {"2", {"D", "E"}},
        {"12", {"X"}},
        {"3", {"P", "Q"}}};

    vector<string> curResult;
    printCombinations(str, map, curResult);
}

void printCombinations(string str, unordered_map<string, list<string>>& map, vector<string>& cur
    if (str.empty()) {
        cout << curResult << endl;
        return;
    }

    for (int i = 1; i <= str.size(); i++) {
        string cur = str.substr(0, i);
        string rest = str.substr(i);

        for (auto& e : map[cur]) {
            curResult.push_back(e);
            printCombinations(rest, map, curResult);
            curResult.pop_back();
        }
    }
}
```

# Longest Common Substring

```
int longestCommonSubsequenceDP(string text1, string text2) {
    int n = text1.size();
    int m = text2.size();

    vector<vector<int>> table(n+1, vector<int>(m+1));

    for(int i=0;i<=n;i++) {
        for(int j=0;j<=m;j++) {
            if(i == 0 || j == 0) table[i][j] = 0;
            else if(text1[i-1] == text2[j-1]) table[i][j] = table[i-1][j-1] + 1;
            else table[i][j] = max(table[i-1][j], table[i][j-1]);
        }
    }

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

```
int longestCommonSubsequenceRecursive(string text1, int n, string text2, int m) {
    if(n == 0 || m == 0) return 0;

    if(text1[n-1] == text2[m-1]) return 1 + longestCommonSubsequenceRecursive(text1, n-1, text2,

    return max(longestCommonSubsequenceRecursive(text1, n-1, text2, m),
                longestCommonSubsequenceRecursive(text1, n, text2, m-1));
}
```

# Shuffle Array

```
void shuffle(vector<int>& arr) {
    int n = arr.size();
    for (int i = n - 1; i >= 0; i--) {
        int r = rand() % (i + 1);
        swap(arr[i], arr[r]);
    }
}
```

# Shortest common subsequence

```
Given two strings str1 and str2, return the shortest string that has both str1 and str2 as subsequences. If there
Input: str1 = "abac", str2 = "cab"
Output: "cabac"
```

```
string shortestCommonSupersequence(string str1, string str2) {
    string lcs = getLCS(str1, str2);

    // combine all mismatches with lcs string
    string result;
    int i = 0, j = 0;
    for(auto& ch : lcs) {
        while(str1[i] != ch) result += str1[i++];
        while(str2[j] != ch) result += str2[j++];
        result += ch;
        i++,j++;
    }

    return result + str1.substr(i) + str2.substr(j);
}

string getLCS(string& s1, string& s2) {
    int n = s1.length();
    int m = s2.length();

    // get the lcs length
    vector<vector<int>> table(n+1, vector<int>(m+1));
    for(int i=0;i<=n;i++) {
        for(int j=0;j<=m;j++) {
            if(i == 0 || j == 0) table[i][j] = 0;
            else if(s1[i-1] == s2[j-1]) table[i][j] = 1 + table[i-1][j-1];
            else table[i][j] = max(table[i-1][j], table[i][j-1]);
        }
    }

    // determine the actual lcs string
    string result;
    for(int i=n,j=m;i>0&&j>0;) {
        if(s1[i-1] == s2[j-1]) {
            result += s1[i-1];
            i--;
            j--;
        } else if(table[i-1][j] > table[i][j-1]) {
            i--;
        } else {
            j--;
        }
    }

    reverse(result.begin(), result.end());

    return result;
}
```

# Minimum distance between 2 strings

```
Given two strings word1 and word2, return the minimum number of steps required to make word1 and word2 the same.

In one step, you can delete exactly one character in either string.

Input: word1 = "sea", word2 = "eat"
Output: 2
Explanation: You need one step to make "sea" to "ea" and another step to make "eat" to "ea".
```

```
int minDistance(string word1, string word2) {
    return minDistanceUsingLCS(word1, word2);
}

int minDistanceUsingEditDistance(string word1, string word2) {
    int n = word1.length(), m = word2.length();
    vector<vector<int>> table(n+1,vector<int>(m+1));

    for(int i=0;i<=n;i++) {
        for(int j=0;j<=m;j++) {
            if(i == 0) table[i][j] = j;
            else if(j == 0) table[i][j] = i;
            else if(word1[i-1] == word2[j-1]) table[i][j] = table[i-1][j-1];
            else table[i][j] = 1+min(table[i-1][j], table[i][j-1]);
        }
    }

    return table[n][m];
}

int minDistanceUsingLCS(string word1, string word2) {
    int n = word1.length(), m = word2.length();
    vector<vector<int>> table(n+1,vector<int>(m+1));

    for(int i=0;i<=n;i++) {
        for(int j=0;j<=m;j++) {
            if(i == 0 || j == 0) table[i][j] = 0;
            else if(word1[i-1] == word2[j-1]) table[i][j] = 1 + table[i-1][j-1];
            else table[i][j] = max(table[i-1][j], table[i][j-1]);
        }
    }

    int l = table[n][m];

    return n+m - 2*l;
}
```

# Longest Palindromic Subsequence

## Finding longest palindromic using recursive method

T = O(2^n)

```
int longestPalinRec(string s) {
    return longestPalinRec(s, 0, s.length() - 1);
}

int longestPalinRec(string& s, int l, int r) {
    if (l > r) return 0;
    if (l == r) return 1;

    if (s[l] == s[r]) return 2 + longestPalinRec(s, l + 1, r - 1);
    return max(longestPalinRec(s, l + 1, r), longestPalinRec(s, l, r - 1));
}
```

## Finding longest palindromic using recursive method with memoization to avoid recursive calls.

T = O(n^2)

```
int longestPalinUsingMemo(string& s) {
    int n = s.length();
    vector<vector<int>> table(n, vector<int>(n, -1));

    return longestPalinUsingMemo(s, 0, n - 1, table);
}

int longestPalinUsingMemo(string& s, int l, int r, vector<vector<int>>& table) {
    if (l > r) return 0;
    if (l == r) return 1;
    if (table[l][r] != -1) return table[l][r];

    if (s[l] == s[r])
        table[l][r] = 2 + longestPalinUsingMemo(s, l + 1, r - 1, table);
    else
        table[l][r] = max(longestPalinUsingMemo(s, l + 1, r, table), longestPalinUsingMemo(s, l,

    return table[l][r];
}
```

## Finding palindromic subsequence using DP to match the sublength

T = O(n^2)

```
int longestPalinSubseqUsingDP1(string& s) {
    int n = s.length();
    vector<vector<int>> table(n, vector<int>(n, 0));
    for (int i = 0; i < n; i++) table[i][i] = 1;

    for (int l = 2; l <= n; l++) {
        for (int i = 0; i < n - l + 1; i++) {
            int j = i + l - 1;
            if (l == 2 && s[i] == s[j])
                table[i][j] = 2;
            else if (s[i] == s[j])
                table[i][j] = table[i + 1][j - 1] + 2;
            else
                table[i][j] = max(table[i + 1][j], table[i][j - 1]);
        }
    }

    return table[0][n - 1];
}
```

## Using LCS DP to get the common subsequence between string and its reverse.

T = O(n^2)

```
int longestPalinSubseqUsingLCSDP(string& s1) {
    int n = s1.length();
    string s2 = s1;
    reverse(s2.begin(), s2.end());
    vector<vector<int>> table(n + 1, vector<int>(n + 1));

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                table[i][j] = 0;
            else if (s1[i - 1] == s2[j - 1])
                table[i][j] = 1 + table[i - 1][j - 1];
            else
                table[i][j] = max(table[i - 1][j], table[i][j - 1]);
        }
    }

    return table[n][n];
}
```

**Above methods only finds the length of longest palindromic subsequence but not actually prints. Following methods prints**

### it using LCS DP and traverse the table from back and follow the track in reverse direction.

```cpp
void printLongestPalindromicSubsequence(string& s1) {
    string s2(s1.rbegin(), s1.rend());

    int n = s1.length();

    vector<vector<int>> table(n + 1, vector<int>(n + 1));

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                table[i][j] = 0;
            else if (s1[i - 1] == s2[j - 1])
                table[i][j] = 1 + table[i - 1][j - 1];
            else
                table[i][j] = max(table[i - 1][j], table[i][j - 1]);
        }
    }

    string result;
    for (int i = n, j = n; i > 0 && j > 0;) {
        if (s1[i - 1] == s2[j - 1]) {
            result += s1[i - 1];
            i--, j--;
        } else if (table[i - 1][j] > table[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }

    reverse(result.begin(), result.end());

    cout << "Result: " << result << endl;
}
```

# Count subarrays with elements less that K.

```
int countSubarrays(vector<int>& arr, int k) {
    int l = 0;
    int n = arr.length();
    int cnt = 0;
    for (int r = 0; r < n; r++) {
        if (arr[r] < k)
            cnt += (r - l + 1);
        else
            l = r + 1;
    }

    return cnt;
}
```

# Merge overlapping intervals

```
void mergeIntervalsInPlace2(vector<Interval>& intervals) {
    sort(intervals.begin(), intervals.end(), [](auto& f, auto& s) { return f.start < s.start; })

    vector<Interval> result;
    for (auto& interval : intervals) {
        if (result.empty() || result.back().end < interval.start) {
            result.push_back(interval);
        }
        else {
            result.back()[1] = max(result.back().end, interval.end);
        }
    }

    return result;
}
```

```
void mergeIntervalsInPlace(vector<Interval>& intervals) {
    int i = 0;
    for (int j = 1; j < intervals.size(); j++) {
        if (overlap(intervals[i], intervals[j])) {
            merge(intervals[i], intervals[j]);
        } else {
            i++;
            intervals[i] = intervals[j];
        }
    }

    while (intervals.size() > i + 1) intervals.pop_back();
}

bool overlap(Interval it1, Interval it2) {
    return it2.start <= it1.end && it2.start >= it1.start;
}

void merge(Interval& it1, Interval& it2) {
    it1 = {min(it1.start, it2.start), max(it1.end, it2.end)};
}
```

# Merge 2 sorted arrays

```
vector<Interval> mergeArrays(vector<Interval>& arr1, vector<Interval>& arr2) {
    vector<Interval> output;
    int i = 0, j = 0;

    while (i < arr1.size() && j < arr2.size()) {
        if (arr1[i].start <= arr2[j].start) {
            insertOrMerge(output, arr1[i]);
            i++;
        } else {
            insertOrMerge(output, arr2[j]);
            j++;
        }
    }

    while (i < arr1.size()) insertOrMerge(output, arr1[i++]);
    while (j < arr2.size()) insertOrMerge(output, arr2[j++]);

    return output;
}

void insertOrMerge(vector<Interval>& out, Interval it) {
    if (out.empty() || it.start > out.back().end) {
        out.push_back(it);
    } else {
        auto cur = out.back();
        out.pop_back();

        cur = {min(cur.start, it.start), max(cur.end, it.end)};
        out.push_back(cur);
    }
}
```

# Count Islands in a matrix with 1s and 0s where 1 represent land and 0 represent water.

- An island a continous land of 1s (moving up, down, left, right)

*****Using DFS**

- We can mark the matrix itself to mark the visited cells. If that's not allowed, we can use seperate visited matrix for that.

```
int countIslands(vector<vector<int>>& matrix, int n, int m) {
    for(int i=0;i<n;i++) {
        for(int j=0;j<m;j++) {
            if(matrix[i][j] == 1) {
                dfs(matrix, i, j);
                cnt++;
            }
        }
    }

    return cnt;
}
```

- we can use bfs as well instead of dfs in above approach.

***Count without dfs by looking up or left and increase islands count.

```
int countIslands(vector<vector<int>>& matrix, int n, int m) {
    int cnt = 0;
    for(int i=0;i<n;i++) {
        for(int j=0;j<m;j++) {
            if(matrix[i][j] ==1) {
                if((i == 0 || matrix[i-1][j] == 0) &&
                    (j == 0 || matrix[i][j-1] == 0))
                {
                    cnt++;
                }
            }
        }
    }

    return cnt;
}
```

# Longest increasing subsequence

***Using recursion***

```
int longestUsingRecursion(vector<int>& arr) {
    return longestUsingRecursion(arr, INT_MIN, 0, arr.size());
}

int longestUsingRecursion(vector<int>& arr, int mn, int l, int n) {
    if (l == n) return 0;

    int l1 = 0;
    if (arr[l] > mn) l1 = 1 + longestUsingRecursion(arr, arr[l], l + 1, n);
    int l2 = longestUsingRecursion(arr, mn, l + 1, n);

    return max(l1, l2);
}
```

### *Using dynamic programming*

```
int longestUsingDP(vector<int>& arr) {
    int n = arr.size();
    vector<int> table(n, 0);
    table[0] = 1;

    for (int i = 1; i < n; i++) {
        for (int j = i - 1; j >= 0; j--) {
            if (arr[i] > arr[j])
                table[i] = max(table[i], table[j] + 1);
            else
                table[i] = 1;
        }
    }

    return table[n - 1];
}
```