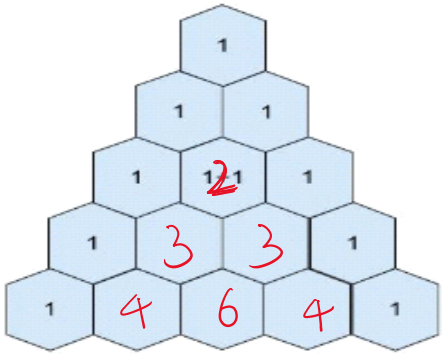
Pascal's triangle:

Variation 1: given row 'r' & col 'c', print the element at pos(r,c) in Pascal Triangle

Variation 2: given the row number n, print the n-th row of Pascal's triangle

Variation 3: given the number of row n, print the first n rows of Pascal's triangle.



// Variation 1:

- brute force approach is to generate the entire the triangle & return the element

- Better way is using using the combinations formula.

given, r = rowNum
c = colNum

$$\Rightarrow \binom{r-1}{c-1}$$

- calculating combination: $^{n}C_{r} = \frac{n!}{r! \times (n-r)!}$

Eg: $^{7}C_{3} = \frac{7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}{(3 \times 2 \times 1) \times (4 \times 3 \times 2 \times 1)} = 21$

```
func nCr(n, r){
    int res = 1;
    for(i=0 ⟶ r){
        res = res * (n-i);
        res = res / (i+1);
    }
    return res;
}
```

time complexity: $O(r)$
space complexity: $O(1)$

```
rowNum = r
colNum = c
ele at (r,c) = nCr(r-1, c-1);
```

// Variation 2:

- Nth row would contain n no. of elements.
- every element has the formula of ${}^{r-1}C_{c-1}$ r = rowNum, c = colNum
- run a loop n times and pass the values in the function nCr(n, r);

```
vector<int> row;
for(c=1; c<=n; c++)
    row.push_back(nCr(n-1, c-1));
```

$O(n * r)$

- if we observe there is an optimized method.

	${}^5C_0$	${}^5C_1$	${}^5C_2$	${}^5C_3$	${}^5C_4$	${}^5C_5$
6th row:	1	5	10	10	5	1
		$\frac{5}{1}$	$\frac{5\times4}{1\times2}$	$\frac{5\times4\times3}{1\times2\times3}$	$\frac{5\times4\times3\times2}{1\times2\times3\times4}$	
		prev				
		prev *= $\frac{4}{2}$	prev *= $\frac{3}{3}$	prev *= $\frac{2}{4}$		

formula: $ans = ans * \frac{(row - col)}{col}$

(col is 0 based indexing)

```
getRow(rowNum){
    vector<int> row;
    int ele = 1;
    row.pushback(ele);
    for(i=1 ⟶ rowNum){
        ele *= (rowNum - i);
        ele /= i;
        row.push_back(ele);
    }
    return row;
}
```

time complexity: O(n)
space complexity: O(1)

// Variation 3: • use the above method to get rows and a loop to get first n rows.

```
getTriangle(numRows){
    vector<vector<int>> triangle;
    for(i=1; i<=numRows; i++)
        triangle.pushback(getRow(i));

    return triangle;
}
```

time complexity: $O(n^2)$
space complexity: O(1)

#229. Majority Element - II

Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.

From <https://leetcode.com/problems/majority-element-ii/description/>

[3,2,3] >> [3] [1] >> [1] [1,2,3] >> [1,2]

// Brute force :

- at max there will two elements appearing more than $\frac{n}{3}$ times.
- use an outer loop to select the element, initialize cnt = 1 inside the loop.
- use inner loop to count the occurence of the curr. element, if (a[j] == a[i]) cnt ++;
- outside the inner loop, check:
  if (cnt > $\frac{n}{3}$) ans.pushback(a[i]);
- also keep a if (ans.size == 2) break;

outside the outer loop return the ans;

```
for(i=0 ——————> n){
   if (ans.size() == 0 || ans[0] != nums[i]){
      cnt = 0;
      for(j = 0 ——————> n){
         if (nums[j] == nums[i]) cnt++;
      }
      if (cnt > n/3) ans.push(nums[i]);
   }
   if (ans.size() == 2) break;
}
return ans;
```

time complexity: $O(n^2)$

return ans;

time complexity: $O(n^2)$

space complexity: $O(1)$

// Better Solution: (hashing)

- declare a hash map to store (element, occurrence) of each element.
- iterate over the array and count the occurrences of each element as: `hash[nums[i]]++;`
- iterate over the hash, if `hash[i] > n/3`, add to ans, if `ans.size() == 2`, break and return ans.

```
map<int, int> hash;
for(i = 0 ⟶ n){
    hash[nums[i]]++;
    if(hash[nums[i]] > n/3)
        ans.push_back(nums[i]);
    if(ans.size() == 2) break;
}
return ans;
```

time complexity: O(n)
space complexity: O(n)

// Optimal Solution: (Moore's Voting Algorithm (modified))

- we will replicate the Moore's voting algorithm from the problem $(\frac{n}{2})$ times, but with a slite modification.

- Moore's voting algorithm works on the intuition that if cnt = 0, make curr. ele as major_Ele, and cnt++, - if major_Ele appears again cnt++ else, cnt--.

- Try to do the dam for two elements : ele 1, cnt 1 = 0; ele 2, cnt 2 = 0

- Now apply the same operations discussed above.

```
cnt1 = 0, cnt2 = 0
ele1 = INT_MIN, ele2 = INT_MIN;
for(i=0 ⟶ n){
    if(cnt1==0 && nums[i] != ele2){
        ele1 = nums[i];
        cnt1++;
    else if(cnt2==0 && nums[i] != ele1){
        ele2 = nums[i];
        cnt2++;
    else if(nums[i] == ele1) cnt1++;
    else if(nums[i] == ele2) cnt2++;
    else cnt1--; cnt2--;

}
 cnt1 = 0, cnt2 = 0;
for(i=0 ⟶ n){
    if(nums[i] == ele1) cnt1++;
    else (nums[i] == ele2) cnt2++;
}
mini = (int)(n/3)+1;
if(cnt1 >= mini) ans.push_back(ele1);
if(cnt2 >= mini) ans.push_back(ele2);

return ans;

}
```

time complexity : O(n)
space complexity : O(1)

#15. 3Sum

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.
Notice that the solution set must not contain duplicate triplets.

From <https://leetcode.com/problems/3sum/description/>

[-1,0,1,2,-1,-4] // [[-1,-1,2],[-1,0,1]]

[0,1,1] // []

[0,0,0] // [[0,0,0]]

// Brute force: (using three loops)

- find all the triplets possible, and select the ones which sum upto 0.

- first loop to select first element,
- second loop to select second element
- third loop to select third element
- sum up all elements, if sum == 0, store them,
- sort the elements, and store them in a set
- copy the set into the vector<vector<int>> ans;

```
set<vector<int>> st;
for(i=0 ——→ n){
    for(j=i+1 ——→ n){
        for(k=j+1 ——→ n){
            if(nums[i] + nums[j] + nums[k] == 0){
                vector<int> temp={nums[i], nums[j], nums[k]};
                sort(temp.begin(), temp.end());
                st.insert(temp);
            }
        }
    }
}
return set copied into vector<vector<int>>;
```

// Better Solution: (Using two loops and hashing)

- nums[i] + nums[j] + nums[k] = 0
  ⇒ nums[k] = −(nums[i] + nums[j])
  ↳ search this in the array using hashing
  ↳ while searching make sure that the element to be searched is not one of (nums[i] or nums[j])
- to avoid the duplication do this: we maintain an range of (i → j)

arr = [−1, 0, 1, 2, −1, −4]

i j j j j j

5
−1
2
1
0

rem = −(−1 + 0) = +1
rem = −(−1 + 1) = 0
rem = −(−1 + 2) = −1
rem = −(−1 − 1) = 2
rem = −(−1 − 4) = 5

{(−1, 1, 0), (−1, −1, 2)}

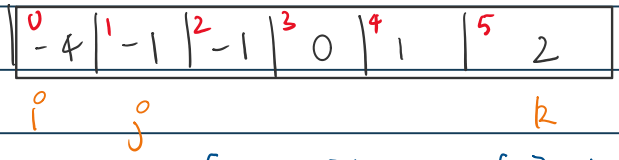
perform all these steps for (i = 0 → n), and after each iteration of i clear the map.

```
set<vector<int>> st;
for(i=0 ⟶ n){
    set<int> hash;
    for(j = i+1 ⟶ n){
        if(hash.find(-(a[i]+a[j])) != hash.end()){
            vector<int> temp={a[i],a[j],-(a[i]+a[j])};
            sort(temp.begin(), temp.end());
            st.insert(temp);
        }
        hash.insert(a[j]);
    }
}
return {st.begin(), st.end()};
```

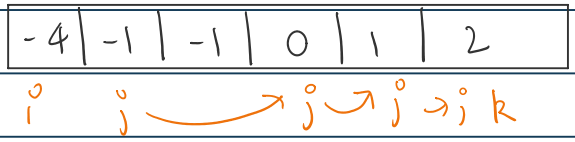
time complexity: $O(n^2)$
space complexity: $O(n)$

// Optimal Solution: (two pointer approach)

- sort the array.
- now take three pointers i, j and k assuming
  i -> left, j -> mid, k -> right,
- fix the i pointer to the left most position, j = i+1 & k to the right most



- find sum of a[i], a[j] & a[k]:
  check: if (sum == 0) add to ans, & move j & k
  else if (sum < 0) move j until j moves to next different element
  else move k to left until k is at next different element
- do these operations until j < k, and after each iteration of move 'i' to next different element.
- Do until i < n-2.



-4 + -1 + 2 = -3 < 0, move j
-4 + 0 + 2 = -2 < 0, move j
-4 + 1 + 2 = -1 < 0, move j EXIT next iteration.

-1 -1 + 2 = 0, move j, k
-1 + 0 + 1 = 0, move j, k
EXIT, next iteration.

{(-1, -1, 2)}
{(-1, -1, 2), (-1, 0, 1)}

Continue the iteration of 'i' until i < n-2.

```
vector<vector<int>> ans;
sort(nums);
for(i=0 ------> n-2){
    if(i>0 && nums[i]==nums[i-1]) continue;
      j= i+1;
      k=n-1;
      while(j<k){
       sum = nums[i]+nums[j]+nums[k];
       if(sum==0){
        ans.push({nums[i],nums[j],nums[k]});
          j++; k--;
       while(j<k && nums[j]==nums[j-1]) j++;
       while(j<k && nums[k]==nums[k+1] k--;
       }
       else if(sum<0) j++;
       else k--;
      }
}
return ans;
```

time complexity: $O(n\log n) + O(n^2)$

space complexity: $O(ans)$

18. 4Sum

Given an array nums of n integers, return *an array of all the* ***unique*** *quadruplets* [nums[a], nums[b], nums[c], nums[d]] such that:

- 0 <= a, b, c, d < n
- a, b, c, and d are **distinct**.
- nums[a] + nums[b] + nums[c] + nums[d] == target

You may return the answer in **any order**.

Input: nums = [1,0,-1,0,-2,2], target = 0
Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

// Brute force (using four loops)

- find all the quadruples and sum them, if sum == target, push it into ans.

```
for(i=0 ⟶ n){
    for(j=i+1 ⟶ n){
        for(k=j+1 ⟶ n){
            for(l=k+1 ⟶ n){
                sum = a[i]+a[j]+a[k]+a[l];
                if(sum == target){
                temp = {a[i], a[j], a[k], a[l]};
                sort(temp);
                st.insert(temp);
            }
        }
    }
}
return {st.begin(), st.end()};
```

time complexity : $O(n^4)$
space complexity : $O(n^2)$

// Better Solution: (using hash)

- $a[i] + a[j] + a[k] + a[l] = target$

  $\Rightarrow a[l] = target - (a[i] + a[j] + a[k])$

  ↳ try to search for this in the using hashing

  ↳ make sure the value you are searching for none of these $(a[i], a[j], a[k])$

- we are gonna run two loop(i) & loop(j), in the third loop (k) we will do following operations:

0	1	2	3	4	5
1	0	-1	0	-2	2

i j k k k

$t = 0$

$0 - (1 + 0 - 1) = 0$ ✗

$0 - (1 + 0 + 0) = -1$ ✗

$0 - (1 + 0 - 2) = 1$ ✗

$0 - (1 + 0 + 2) = -3$ ✗

2
-2
0
-1
0

1	0	-1	0	-2	2

i j k k

$t = 0$

$0 - (1 - 1 + 0) = 0$ ✓ $(-1, 0, 0, 1)$

$0 - (1 - 1 - 2) = 2$ ✗

$0 - (1 - 1 + 2) = -2$ ✗

2
-2
0

- continue all these step until the loop(i) & loop(j) go out of bound.
  - make sure to empty the for each iteration j.

```
for(i=0 ⟶ n){
  for(j=i+1 ⟶ n){
    set<>.. hash;
    for(k=j+1 ⟶ n){
      sum = a[i] + a[j] + a[k]
      fourth = target - sum;
      if(hash.find(fourth) != hash.end()){
      temp={ a[i], a[j], a[k], (int) fourth};
      sort(temp);
       st.insert(temp);
      }
      hash.insert(nums[k]);
    }
  }
}
return {st.begin, st.end()};
```

time complexity:	$O(n^3)$
space complexity:	$O(n^2)$

// Optimal Solution : (two pointer approach)

- to get rid of set, we will sort the array to get only unique quadruples.
- we will have two loops, loop(i) & loop(j), we will fix them at i=0 & j=i+1,
- take two pointers l=j+1, h=n-1, now until l < h do following operations:

```
sum = a[i]
sum += a[j] + a[l] + a[h];
```

if(sum == target) : insert quadruples in ans.

```
l++, h--;
move (l to right until a[l] == a[l-1])
move (h to left until a[h] == a[h+1])
else if (sum < target) l++
  else h--;
```

- after iteration move (j to right until a[j] == a[j-1]
- after iteration of i do the same for ii
- Outside the loop return ans.

```
vector<vector> ans;
sort(a);
for(i=0 ⟶ n){
  if(n>0 && a[i]==a[i-1]) continue;
  for(j=i+1 ⟶ n){
    if(j!=i+1 && a[j]==a[j-1]) continue;
      l = j+1, h = n-1;
      while (l < h){
      sum = a[i] + a[j] + a[l] + a[h];
      if(sum == target){
      ans.push({a[i], a[j], a[l], a[h]});
       l++; h--;
      while (l<h && a[l]==a[l-1]) l++
      while (l<h && a[h] == a[h+1]) h--;
      }
      else if (sum < target) l++;
      else h--;
    }
  }
}
return ans;
```

time complexity : $O(n^3)$

space complexity : $O(n^2)$ ⇓ to store ans

Longest subarray with zero Sum:

Given an array containing both positive and negative integers, we have to find the length of the longest subarray with the sum of all elements equal to zero.

[9, -3, 3, -1, 6, -5] >> 5
[6, -2, 2, -8, 1, 7, 4, -10] >> 8

Brute force: (using two loops)

- Find all the subarrays with sum = 0, return the length of longest substring.
- use two loops, loop(i) to select the starting point, loop(j) to select the ending point.
- calculate sum, if (sum == 0)
  update maxi = max(maxi, j-i+1);
- return the maxi as ans outside the loop.

```
maxi = 0;
for (i=0 ⟶ n){
  for(j=i ⟶ n){
    sum += a[i];
    if(sum == 0)
      maxi = max(maxi, j-i+1);
  }
}
return maxi;
```

time complexity: $O(n^2)$
space complexity: $O(1)$

// Better Solution : (using map)

- Suppose, sum(i, j) = S, and sum(x, j) = S
  ⇒ sum(x+1, j) = 0 (i < x < j)
- declare a map to store (sum, index)
- traverse the array, and calculate sum, and check
  (sum == 0) : maxi = i+1
  (sum != 0) : check the map for sum
  if(sum → map) : update maxi = max(maxi, i - mp[sum])
  else : insert (sum, i) in map

```
maxi = 0, sum = 0
map<int, int> mp;
for(i = 0 ------> n){
    sum += a[i];
    if(sum == 0)
        maxi = i+1
    else{
        if(mp.find(sum) != mp.end())
            maxi = max(maxi, i - mp[sum]);
        else mp[sum] = i;
    }
}
return maxi;
```

time complexity : O(n)
space complexity : O(n)

Subarray with given XOR :

Given an array of integers A and an integer k. Find the total number of subarrays having bitwise XOR of all elements equal to k.

[4, 2, 2, 6, 4], k = 6 >> [4,2], [4, 2, 2, 6, 4], [2, 2, 6], [6]

[5, 6, 7, 8, 9], k = 5 >> [5], [5, 6, 7, 8, 9] = 2

// Brute force (Using two loops)

- use two loops to generate all the subarrays and calculate the xorr, if xorr == k, cnt++ and return the cnt as ans.
- first loop(i) to select an element, loop(j) to find all subarrays starting with (i).
- Calculate xorr of subarray (i → j), if xorr == k, increment cnt else move to next subarray.

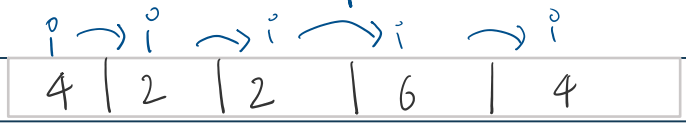
```
cnt = 0;
for(i=0 ⟶ n){
    xorr = 0;
    for(j=i ⟶ n){
        xorr = xorr ^ a[j];
        if(xorr == k)
            cnt++;
    }
}
return cnt;
```

time complexity: $O(n^2)$
space complexity: $O(1)$

// Better Solution : (using map)

- suppose xorr of subarray $(0, i) = x_f$
  xorr of subarray $(j, i) = k$
  $\Rightarrow$ xorr of subarray $(0, j) = x_f \wedge k$

- Declare a map to store the XORs and their cnt.
- set the value of 0 as 1 in map
- run a loop over the array
- inside the loop, calculate the xorr by
  - XOR the current the element to prefix XOR
  - calculate the prefix XOR i.e $x_f \wedge k$
  - add the occurrence of $x_f \wedge k$ to the cnt
  - store the current prefix x_f, in the map & increment it by 1.
- outside the loop return cnt



xr : 0^4=4 4^2=6 6^2=4 4^6=2 2^4=6

x : 4^6=2 6^6=0 4^6=2 2^6=4 6^6=0

x ✗ ✓ ✗ ✓

cnt = 0 1 2 3 4 (+1 +2 +1)

(2,1
(6,2
(4,2
(0,1

```
cnt=0, xorr=0, map<int, int> mp;
mp[0] ++;
for(i=0 ⟶ n){
    xorr = xorr ^ a[i];
    x = xorr ^ k;
    cnt += mp[x];
    mp[xorr] ++;
}
return cnt;
```

time complexity: O(n)
space complexity: O(n)