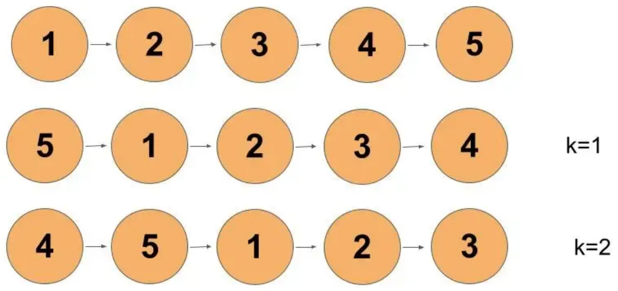
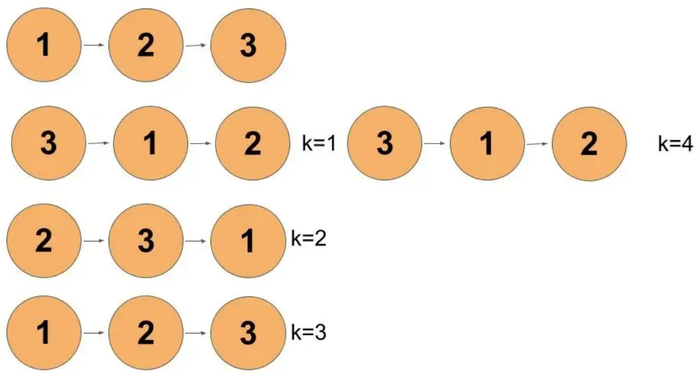
Rotate a Linked List

In this article we will solve the problem: “Rotate a Linked List”
Problem Statement: Given the head of a linked list, rotate the list to the right by k places.
Examples:

```
Example 1:
Input:
     head = [1,2,3,4,5]
     k = 2
Output:
 head = [4,5,1,2,3]
Explanation:
 We have to rotate the list to the right twice.
```



```
Example 2:
Input:
     head = [1,2,3]
     k = 4
Output:
head = [3,1,2]
Explanation:
```

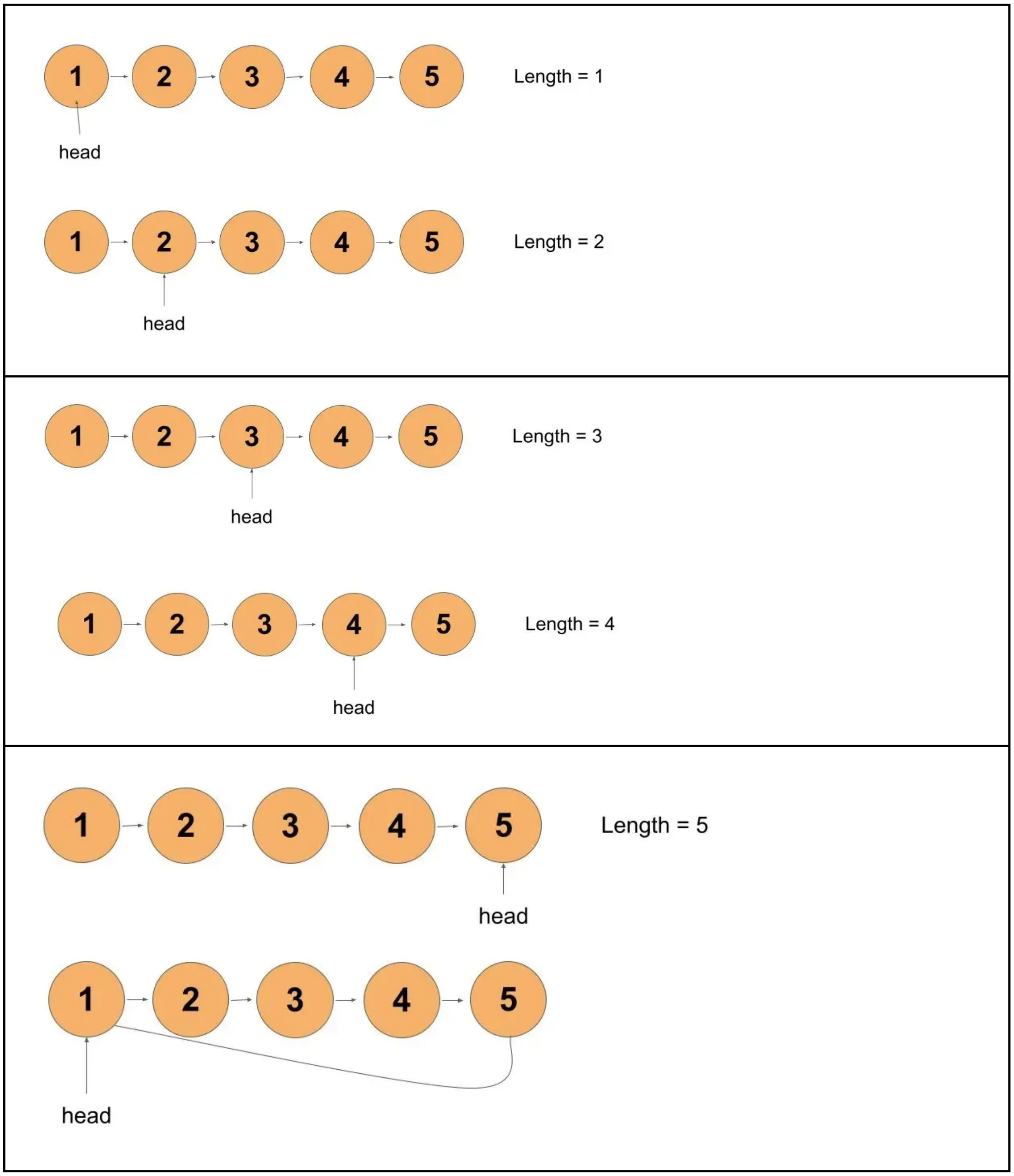


Solution: Brute Force
Approach:
We have to move the last element to first for each k.
For each k, find the last element from the list. Move it to the first.
Code:
```
//utility function to rotate list by k times
node* rotateRight(node* head,int k) {
    if(head == NULL||head->next == NULL) return head;
    for(int i=0;i<k;i++) {
        node* temp = head;
        while(temp->next->next != NULL) temp = temp->next;
        node* end = temp->next;
        temp->next = NULL;
        end->next = head;
        head = end;
    }
    return head;
}
```
Output:
Original list: 1->2->3->4->5
After 2 iterations: 4->5->1->2->3
Time Complexity: O(Number of nodes present in the list*k)
Reason: For k times, we are iterating through the entire list to get the last element and move it to first.
Space Complexity: O(1)
Reason: No extra data structures is used for computations

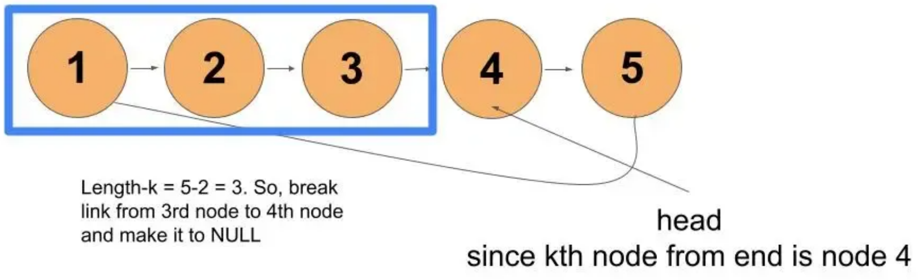
Solution: Optimal Solution
Approach:
Let's take an example.
head = [1,2,3,4,5] k = 2000000000
If we see brute force approach, it will take O(5*2000000000) which is not a good time complexity when we can optimize it.
We can see that for every k which is multiple of the length of the list, we get back the original list. Try to operate brute force on any linked list for k as multiple of the length of the list.
This gives us a hint that for k greater than the length of the list, we have to rotate the list for k%length of the list. This reduces our time complexity.
Steps to the algorithm:-
- Calculate length of the list
- Connect the last node to the first node, converting it to a circular linked list.
- Iterate to cut the link of the last node and start a node of k%length of list rotated list.

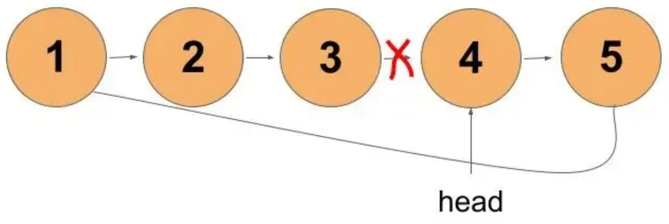
Dry Run:

Let’s calculate the length of the list by iterating on it until it reaches null and increasing the count. Once the length is calculated we will connect the last node to the first node.



Now, the length of the list is 5 and k is 2. k is less than the length of the given list. So, we will have the head of the rotating list at the kth element from the end remove the link from the length-k node from its next node and make it NULL.


1
2
3
4
5
Length-k = 5-2 = 3. So, break link from 3rd node to 4th node and make it to NULL
head
since kth node from end is node 4



Thus, we received our desired output.

Code:

```
//utility function to rotate list by k times
node* rotateRight(node* head,int k) {
    if(head == NULL||head->next == NULL||k == 0) return head;
    //calculating length
    node* temp = head;
    int length = 1;
    while(temp->next != NULL) {
        ++length;
        temp = temp->next;
    }
    //link last node to first node
    temp->next = head;
    k = k%length; //when k is more than length of list
    int end = length-k; //to get end of the list
    while(end--) temp = temp->next;
    //breaking last node link and pointing to NULL
    head = temp->next;
    temp->next = NULL;

    return head;
}
```

Output:

Original list: 1->2->3->4->5

After 2 iterations: 4->5->1->2->3

Time Complexity: O(length of list) + O(length of list – (length of list%k))

Reason: O(length of the list) for calculating the length of the list. O(length of the list – (length of list%k)) for breaking link.
Space Complexity: O(1)
Reason: No extra data structure is used for computation.

3 Sum : Find triplets that add up to a zero

Problem Statement: Given an array of N integers, your task is to find unique triplets that add up to give sum zero. In short, you need to return *an array of all the unique* triplets [arr[a], arr[b], arr[c]] such that i!=j, j!=k, k!=i, and their sum is equal to zero.
Examples:

```
Example 1:

Input: nums = [-1,0,1,2,-1,-4]

Output: [[-1,-1,2],[-1,0,1]]

Explanation: Out of all possible unique triplets possible, [-1,-1,2]
and [-1,0,1] satisfy the condition of summing up to zero with i!=j!=k

Example 2:

Input: nums=[-1,0,1,0]
Output: Output: [[-1,0,1],[-1,1,0]]
Explanation: Out of all possible unique triplets possible, [-1,0,1]
and [-1,1,0] satisfy the condition of summing up to zero with i!=j!=k
```

Solution:
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1 (Brute Force):
Intuition: While starting out, our aim must be to come up with a working solution first. Thus, given the problem, the most intuitive solution would be using three-pointers and checking for each possible triplet, whether we can satisfy the condition or not.
Approach: We keep three-pointers i,j, and k. For every triplet we find the sum of A[i]+A[j]+A[k]. If this sum is equal to zero, we've found one of the triplets. We add it to our data structure and continue with the rest.

Code:

```
#include<bits/stdc++.h>
using namespace std;
vector < vector < int >> threeSum(vector < int > & nums) {
  vector < vector < int >> ans;
  vector < int > temp;
  int i, j, k;
  for (i = 0; i < nums.size() - 2; i++) {
    for (j = i + 1; j < nums.size() - 1; j++) {
      for (k = j + 1; k < nums.size(); k++) {
        temp.clear();
        if (nums[i] + nums[j] + nums[k] == 0) {
          temp.push_back(nums[i]);
          temp.push_back(nums[j]);
          temp.push_back(nums[k]);
        }
        if (temp.size() != 0)
          ans.push_back(temp);
      }    }  }  return ans;
}
```

Output:

```
The triplets are as follows:
-1 0 1
-1 2 -1
0 1 -1
```

Time Complexity : O(n^3)   // 3 loops

Space Complexity : O(3*k) // k is the no.of triplets

Solution 2 (Optimized Approach):

Intuition : Can we do something better?

I think yes! In our intuitive approach, we were considering all possible triplets. But do we really need to do that? I say we fixed two pointers i and j and came out with a combination of [-1,1,1] which doesn't satisfy our condition. However, we still check for the next combination of say [-1,1,2] which is worthless. (Assuming the k pointer was pointing to 2).

Approach:

We could make use of the fact that we just need to return the triplets and thus could possibly sort the array. This would help us use a modified two-pointer approach to this problem.

Eg) Input : [-1,0,1,2,-1,-4]

After sorting, our array is : [-4,-1,-1,0,1,2]

Notice, we are fixing the i pointer and then applying the traditional 2 pointer approach to check whether the sum of three numbers equals zero. If the sum is less than zero, it indicates our sum is probably too less and we need to increment our j pointer to get a larger sum. On the other hand, if our sum is more than zero, it indicates our sum is probably too large and we need to decrement the k pointer to reduce the sum and bring it closer to zero.

Code:
```
#include<bits/stdc++.h>
using namespace std;
 vector<vector<int>> threeSum(vector<int>& num) {
        vector<vector<int>> res;
        sort(num.begin(), num.end());
        // moves for a
        for (int i = 0; i < (int)(num.size())-2; i++) {
            if (i == 0 || (i > 0 && num[i] != num[i-1])) {
                int lo = i+1, hi = (int)(num.size())-1, sum = 0 -
num[i];
                while (lo < hi) {
                    if (num[lo] + num[hi] == sum) {
                        vector<int> temp;
                        temp.push_back(num[i]);
                        temp.push_back(num[lo]);
                        temp.push_back(num[hi]);
                        res.push_back(temp);
                        while (lo < hi && num[lo] == num[lo+1]) lo++;
                        while (lo < hi && num[hi] == num[hi-1]) hi--;
                        lo++; hi--;
                    }
                    else if (num[lo] + num[hi] < sum) lo++;
                    else hi--;
                }
            }
        }
        return res;
    }
```

Output:
```
The triplets are as follows:
-1 -1 2
-1 0 1
```
Time Complexity : O(n^2)
Space Complexity : O(M).

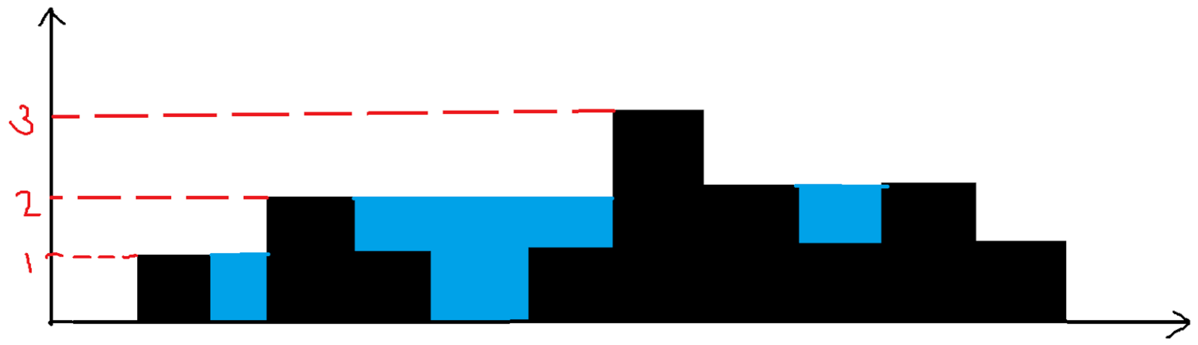
Trapping Rainwater

Problem Statement: Given an array of non-negative integers representation elevation of ground. Your task is to find the water that can be trapped after raining.

Examples:

Example 1:

```
Input: height= [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: As seen from the diagram 1+1+2+1+1=6 unit of water can
be trapped
```



```
Example 2:
Input: [4,2,0,3,2,5]
Output:9
```

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Brute force

Approach: For each index, we have to find the amount of water that can be stored and we have to sum it up.If we observe carefully the amount the water stored at a particular index is the minimum of maximum elevation to the left and right of the index minus the elevation at that index.

Code:

```
#include<bits/stdc++.h>
using namespace std;
int trap(vector < int > & arr) {
  int n = arr.size();
  int waterTrapped = 0;
  for (int i = 0; i < n; i++) {
    int j = i;
    int leftMax = 0, rightMax = 0;
```

```
      while (j >= 0) {
       leftMax = max(leftMax, arr[j]);
       j--;
     }
     j = i;
     while (j < n) {
       rightMax = max(rightMax, arr[j]);
       j++;
     }
     waterTrapped += min(leftMax, rightMax) - arr[i];
   }
   return waterTrapped;
}
int main() {
  vector < int > arr;
  arr = {0,1,0,2,1,0,1,3,2,1,2,1};
  cout << "The water that can be trapped is " << trap(arr) << endl;
}
```

Output: The water that can be trapped is 6
Time Complexity: O(N*N) as for each index we are calculating leftMax and rightMax so it is a nested loop.
Space Complexity: O(1).

Solution 2:Better solution
Intuition: We are taking O(N) for computing leftMax and rightMax at each index. The complexity can be boiled down to O(1) if we precompute the leftMax and rightMax at each index.
Approach: Take 2 array prefix and suffix array and precompute the leftMax and rightMax for each index beforehand.Then use the formula min(prefix[i],suffix[i])-arr[i] to compute water trapped at each index.
Code:

```
#include<bits/stdc++.h>
using namespace std;
int trap(vector < int > & arr) {
  int n = arr.size();
  int prefix[n], suffix[n];
  prefix[0] = arr[0];
  for (int i = 1; i < n; i++) {
    prefix[i] = max(prefix[i - 1], arr[i]);
  }
  suffix[n - 1] = arr[n - 1];
  for (int i = n - 2; i >= 0; i--) {
    suffix[i] = max(suffix[i + 1], arr[i]);
  }
  int waterTrapped = 0;
  for (int i = 0; i < n; i++) {
    waterTrapped += min(prefix[i], suffix[i]) - arr[i];
  }
  return waterTrapped;
}

int main() {
  vector < int > arr;
  arr = {0,1,0,2,1,0,1,3,2,1,2,1};
  cout << "The water that can be trapped is " << trap(arr) << endl;
}
```

Output: The water that can be trapped is 6
Time Complexity:O(3*N) as we are traversing through the array only once. And O(2*N) for computing prefix and suffix array.
Space Complexity:O(N)+O(N) for prefix and suffix arrays.

Solution 3:Optimal Solution(Two pointer approach)
Approach: Take 2 pointer l(left pointer) and r(right pointer) pointing to 0th and (n-1)th index respectively.Take two variables leftMax and rightMax and initialise it to 0.If heigh[l] is less than or equal to height[r] then if leftMax is less then height[l] update leftMax to height[l] else add leftMax-height[l] to your final answer and move the l pointer to the right i.e l++.If height[r] is less then height[l],then now we are dealing with the right block.If height[r] is greater then rightMax,then update rightMax to height[r] else add rightMax-height[r] to the final answer.Now move r to the left. Repeat theses steps till l and r crosses each other.
Intuition: We need a minimum of leftMax and rightMax.So if we take the case when height[l]<=height[r] we increase l++, so we can surely say that there is a block with height more than height[l] to the right of l. And for the same reason when height[r]<=height[l] we can surely say that there is a block to the left of r which is at least of height height[r]. So by traversing with these cases and using two pointers approach the time complexity can be decreased without using extra space.
Code:

```
#include<bits/stdc++.h>
using namespace std;
int trap(vector < int > & height) {
  int n = height.size();
  int left = 0, right = n - 1,res = 0, maxLeft = 0, maxRight = 0;
  while (left <= right) {
    if (height[left] <= height[right]) {
      if (height[left] >= maxLeft) {
        maxLeft = height[left];
      } else {
        res += maxLeft - height[left];
      }
      left++;
    }
else {
      if (height[right] >= maxRight) {
        maxRight = height[right];
      } else {
        res += maxRight - height[right];
      }
      right--;
    }
  }
  return res;
}

int main() {
  vector < int > arr;
  arr = {0,1,0,2,1,0,1,3,2,1,2,1};
  cout << "The water that can be trapped is " << trap(arr) << endl;
}
```

Output: The water that can be trapped is 6
Time Complexity: O(N) because we are using 2 pointer approach.
Space Complexity: O(1) because we are not using anything extra.

Remove Duplicates in-place from Sorted Array

Problem Statement: Given an integer array sorted in non-decreasing order, remove the duplicates in place such that each unique element appears only once. The relative order of the elements should be kept the same.
If there are k elements after removing the duplicates, then the first k elements of the array should hold the final result. It does not matter what you leave beyond the first k elements.
Note: Return k after placing the final result in the first k slots of the array.
Examples:

```
Example 1:

Input: arr[1,1,2,2,2,3,3]

Output: arr[1,2,3,_,_,_,_]

Explanation: Total number of unique elements are 3, i.e[1,2,3] and
Therefore return 3 after assigning [1,2,3] in the beginning of the
array.
```

Example 2:
Input: `arr[1,1,1,2,2,3,3,3,3,4,4]`
Output: `arr[1,2,3,4,_,_,_,_,_,_,_]`
Explanation: `Total number of unique elements are 4, i.e[1,2,3,4] and Therefore return 4 after assigning [1,2,3,4] in the beginning of the array.`

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Brute Force

Intuition: We have to think of a data structure that does not store duplicate elements. So can we use a Hash set? Yes! We can. As we know HashSet only stores unique elements.

Approach:

- Declare a HashSet.
- Run a for loop from starting to the end.
- Put every element of the array in the set.
- Store size of the set in a variable K.
- Now put all elements of the set in the array from the starting of the array.
- Return K.

Code:

```
#include<bits/stdc++.h>
using namespace std;
int removeDuplicates(int arr[]) {
  set < int > set;
  for (int i = 0; i < 7; i++) {
    set.insert(arr[i]);
  }
  int k = set.size();
  int j = 0;
  for (int x: set) {
    arr[j++] = x;
  }
  return k;
}
int main() {
  int arr[] = {1,1,2,2,2,3,3};
  int k = removeDuplicates(arr);
  cout << "The array after removing duplicate elements is " << endl;
  for (int i = 0; i < k; i++) {
    cout << arr[i] << " ";
  }
}
```

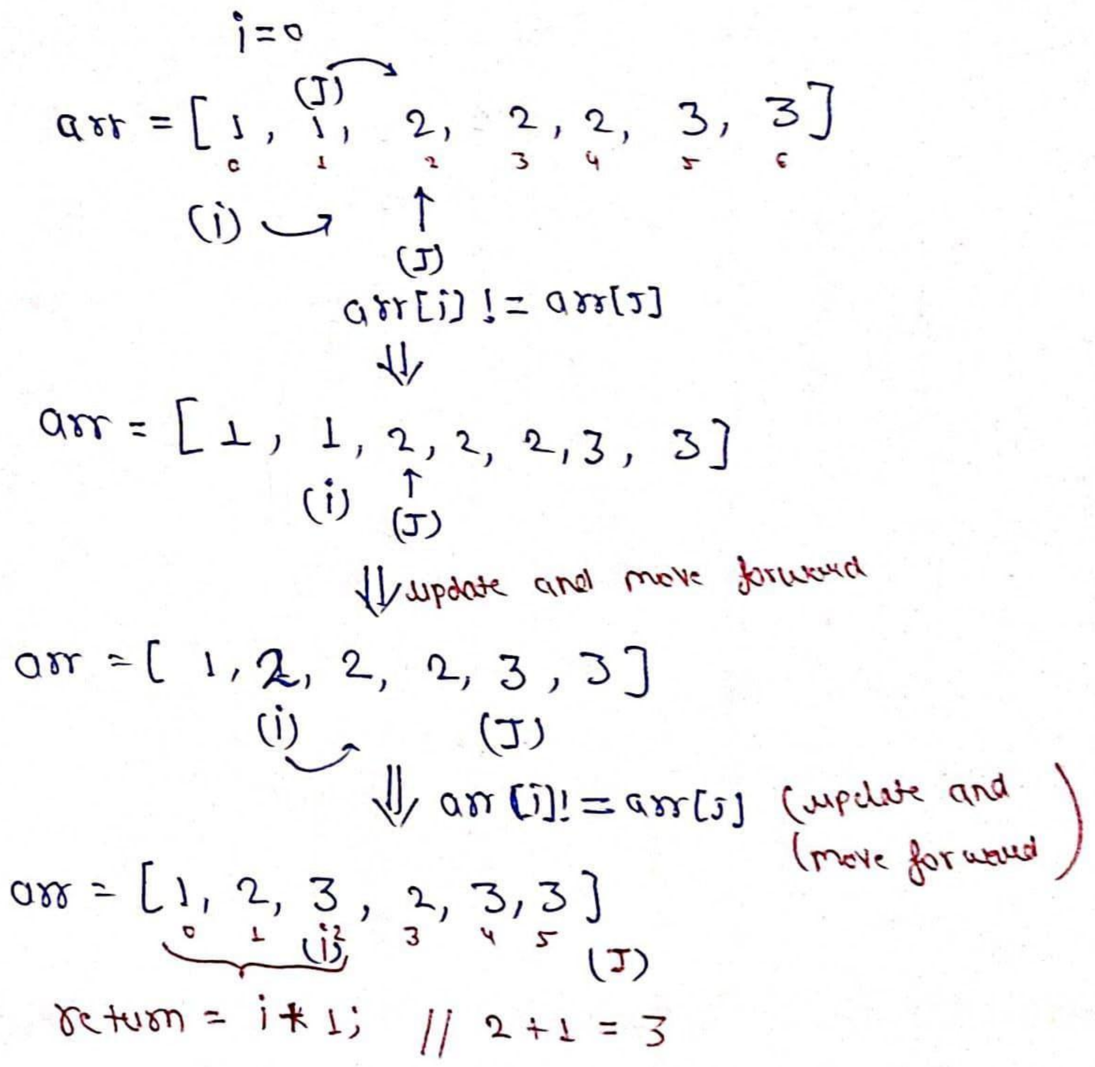
Output: The array after removing duplicate elements is 1 2 3
Time complexity: O(n*log(n))+O(n)
Space Complexity: O(n)

Solution 2: Two pointers

Intuition: We can think of using two pointers ‘i’ and ‘j’, we move ‘j’ till we don’t get a number arr[j] which is different from arr[i]. As we got a unique number we will increase the i pointer and update its value by arr[j].

Approach:

- Take a variable i as 0;
- Use a for loop by using a variable ‘j’ from 1 to length of the array.
- If arr[j] != arr[i], increase ‘i’ and update arr[i] == arr[j].
- After completion of the loop return i+1, i.e size of the array of unique elements.



Code:
```
#include<bits/stdc++.h>
using namespace std;
int removeDuplicates(int arr[]) {
  int i = 0;
  for (int j = 1; j < 7; j++) {
    if (arr[i] != arr[j]) {
      i++;
      arr[i] = arr[j];
    }
  }
  return i + 1;
}
int main() {
  int arr[] = {1,1,2,2,2,3,3};
  int k = removeDuplicates(arr);
  cout << "The array after removing duplicate elements is " << endl;
  for (int i = 0; i < k; i++) {
    cout << arr[i] << " ";
  }
}
```
Output:
```
The array after removing duplicate elements is 1 2 3
```
Time complexity: O(n)
Space Complexity: O(1)

Count Maximum Consecutive One's in the array

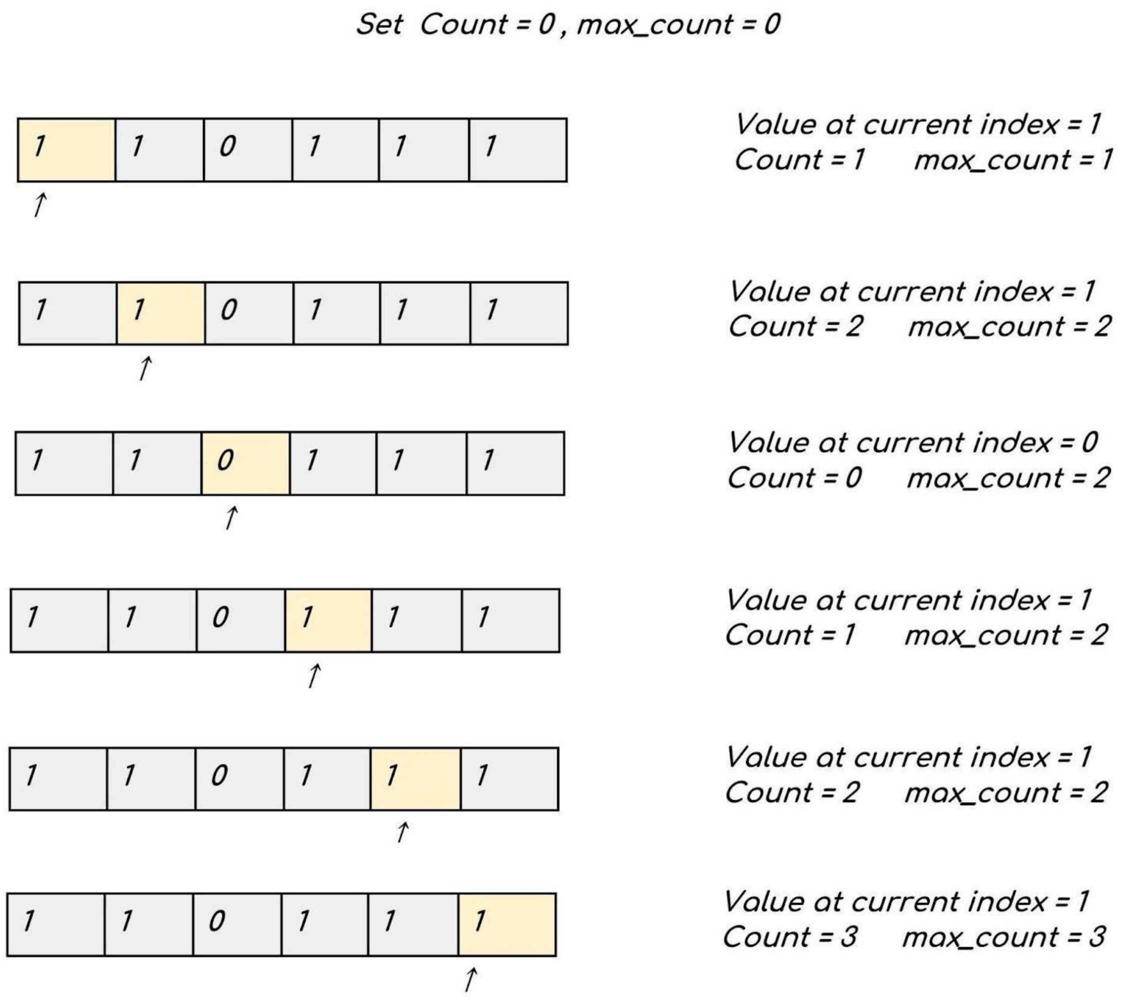
Problem Statement: Given an array that contains **only 1 and 0** return the count of **maximum consecutive** ones in the array.
Examples:
```
Example 1:
Input: prices = {1, 1, 0, 1, 1, 1}
Output: 3
Explanation: There are two consecutive 1's and three consecutive 1's
in the array out of which maximum is 3.
Input: prices = {1, 0, 1, 1, 0, 1}
Output: 2

Explanation: There are two consecutive 1's in the array.
```

Solution:
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*
Approach: We maintain a **variable count** that keeps a track of the number of consecutive 1's while traversing the array. The other variable max_count maintains the maximum number of 1's, in other words, it maintains the answer.
We start traversing from the beginning of the array. Since we can encounter either a 1 or 0 there can be two situations:-

1. If the value at the current index is equal to 1 we **increase the value of count by one.** After updating the count variable if it becomes **more** than the max_count **update the max_count.**
2. If the value at the current index is equal to zero we make the **variable count as 0** since there are **no more consecutive ones**.

See the illustration below for a better understanding


Set Count = 0 , max_count = 0
1 1 0 1 1 1
Value at current index = 1
Count = 1 max_count = 1
Value at current index = 1
Count = 2 max_count = 2
Value at current index = 0
Count = 0 max_count = 2
Value at current index = 1
Count = 1 max_count = 2
Value at current index = 1
Count = 2 max_count = 2
Value at current index = 1
Count = 3 max_count = 3

Code:

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
  public:
    int findMaxConsecutiveOnes(vector < int > & nums) {
      int cnt = 0;
      int maxi = 0;
      for (int i = 0; i < nums.size(); i++) {
        if (nums[i] == 1) {
          cnt++;
        } else {
          cnt = 0;
        }
        maxi = max(maxi, cnt);
      }
      return maxi;
    }
};
int main() {
  vector < int > nums = { 1, 1, 0, 1, 1, 1 };
  Solution obj;
  int ans = obj.findMaxConsecutiveOnes(nums);
  cout << "The maximum  consecutive 1's are " << ans;
  return 0;
}
```

Output: The maximum consecutive 1’s are 3.

Time Complexity: O(N) since the solution involves only a single pass.

Space Complexity: O(1) because no extra space is used.