What is the plan?

Your next 30 days have been divided based on the important topics that you need to know before your next SDE interview. Every day our goal is to solve around 6 most commonly asked questions from that particular topic. By the end of this month, we'll almost cover 180 problems which should give you enough confidence and practice as you go into your online test/interview.
Make sure you go through the previous days' problems every new day so that you don't forget certain concepts.

Prerequisites?

- The basic working of data structures such as Arrays, LinkedLists, Stacks, Queues etc.
- At least one programming language. If you're using C++ you should know how to use STL; mainly vectors, maps, sets, pairs and such. You can refer to this link here https://www.youtube.com/watch?v=g-1Cn3ccwXY. If you're using Java, learn how streams and collections work.
- What are time and space complexities and how to calculate them?
- You can prepare for OS, DBMS and CN here: https://drive.google.com/drive/folders/1Wa7N_plHXc2jUBd9tGBjgMWcRhAl6WZl

DAY 1 (Arrays)

Find the duplicate number in an array of length N. All the numbers in the array vary from 0 to N-1.

(Link to the problem: https://leetcode.com/problems/find-the-duplicate-number/)

Example:
Input: [1,2,2,3]
Output: 2

Approach 1:

You create a new array ***arr*** of length N of type bool and initialize all with false. Now since we know that all the numbers inside the original array, let's call it ***nums***, lie between 0 and N-1. This means, for every number in nums there exists a valid index in arr. For example, 3 is a valid index in arr since it lies between 0 and N-1.
With this knowledge, we can loop through the nums and for every element in nums we change False to True for that numbers matching index in arr, if we encounter a number for which arr is already True, that means that it is our repeating number.

```
int repeating;
for(int i=0;i<N;i++){
    if(arr[nums[i]]){
        repeating=nums[i];
        break;
    }
    arr[nums[i]]=true;
}
```

Time Complexity is O(n)
Space Complexity is O(n) because we are using extra space N for the ***arr*** array.

Now, the way this question gets difficult is by adding a constraint that you are only allowed to use constant auxiliary space or your space complexity is O(1).

Approach 2:

For this approach, we will make use of **Floyd's Algorithm** or otherwise known as the **Hare and Tortoise** approach.
Refer this video for understanding it's working:
https://www.youtube.com/watch?v=LUm2ABqAs1w **(Watch it at 1.5x)**
We will make use of this cycle detection algorithm for solving our problem. Since we already know that all the numbers in the nums array lie between 0 and N-1, hence it in a way forms a closed loop where its entrance or the starting point is the repeating element. Pause and think about it for a minute. Consider all the values in the array as positions and you will never go out of the array.

```
int findDuplicate(vector<int>& nums) {
    int tortoise = nums[0];
    int hare = nums[0];

    while(1){
        tortoise = nums[tortoise];
        hare = nums[nums[hare]];
        if(tortoise==hare) break;
    }
    hare = nums[0];
    while(tortoise!=hare){
        tortoise=nums[tortoise];
        hare=nums[hare];
    }

    return tortoise;
}
```

This will result in a Space Complexity of **O(1)**

Sort an array of 0s, 1s, and 2s.

(Link to the problem: https://leetcode.com/problems/sort-colors/)

Example:
Input: [0,1,0,2,1]
Output: [0,0,1,1,2]

Approach 1:

The easiest way to approach this problem is by counting sort where you can count the number of 0s, 1s and 2s, and replace the existing array according to that.
So for our example:
count_zero = 2
count_one = 2
count_two = 1
Now we can just loop through the existing array and replace the elements accordingly.

The problem with this approach is that it requires two passes, but this problem can be solved with a single pass, hence it's not the best possible solution.

Approach 2:

The concept of this approach is very interesting and simple to understand. If you notice, our main goal is to move all the zeros to the left and all the twos to the right, and if we are able to do so we will automatically get all the ones in the middle. This is also often called **triple partitioning**.

Now, assume that we have a pole in the middle around which everything has to change, in our case that pole is 1.

The way we can approach this problem is by keeping track of the position of 0s and 2s. What do I mean by that? We know that the 0s will always start at the 0th index and the 2s will always end at the last index or N-1, so all we need to do is traverse through the array and whenever we encounter a 0 or a 2, we swap that element with the position where they are originally supposed to exist.
This will become more clear as we write the code.
Assume three variables low=0, check=0 and high = N-1. The variables low and high will keep track of the positions where 0 and 2 should exist and the variable check will traverse through the entire array in search of 0s and 2s.

Let us take this example
nums = [0,1,0,2,1]
Here, initially nums[check] = 0 that means we need to move this element to the index low, the position where 0 should originally be. We now increment check and low by 1.
So our array now looks like [0,1,0,2,1] and low=1 and check=1.

Now, nums[check]=1, but we know that we don't care about 1 because it will come into the correct position as we sort the 0s and the 2s, so we just increment check by 1.

Now, nums[check]=0, and we know that it's original position lies at index low so we swap(nums[check], nums[low]).
And our array looks like [0,0,1,2,1].

We similarly continue this and sort the rest.

```
int low=0,mid=0,hi=nums.size()-1;

    while(mid<=hi){
        switch(nums[mid]){
            case 0:{
                swap(nums[low],nums[mid]);
                low++;
                mid++;
                break;
            }
            case 1:{
                mid++;
                break;
            }
            case 2:{
                swap(nums[hi],nums[mid]);
                hi--;
                break;
            }
        }
    }
```

If you notice, for case 2, we aren't incrementing mid, the reason behind not incrementing mid is that the number that we are swapping with 2 might not necessarily be 1, it can also be 0, so if we swap and increment mid, 0 will get stuck there and won't get back to its original position.

Maximum sum subarray(Kadane's Algorithm)

(Link to the problem: https://leetcode.com/problems/maximum-subarray/)

Example:
Input: [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.

This video explains the algorithm really well
https://www.youtube.com/watch?v=YxuK6A3SvTs

The basic idea behind this algorithm is that we need to ask ourselves a question, whether to add the current element to the current subarray or start a new subarray from the current element. We add the current element to the existing subarray **if it increases the sum**, but if the **current element is larger than the new sum**(subarray sum + current element) then we start a new array.

```
int maxSubArray(vector<int>& nums) {
    int currSum = nums[0];
    int maxSum = nums[0];
    for(int i=1;i<nums.size();i++){
        currSum = max(currSum+nums[i],nums[i]);
        maxSum = max(maxSum,currSum);
    }
    return maxSum;
}
```

Merge two sorted Arrays without extra space

(Link to the problem: https://leetcode.com/problems/merge-sorted-array/)

Example:
Input:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]

For this problem, we are going to make use of 3 pointers. Pointer n1 will point to the end of nums1,i.e. index 2, pointer n2 will point to the end of nums2 and pointer traverse will be used for traversing nums1 from the back and storing values at those indexes.

```
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    int n1=m-1,n2=n-1,traverse=nums1.size()-1;

    while(traverse>=0){

        if(n1<0){
            //We can just replace 'traverse' index with element at n2 in nums2

            nums1[traverse]=nums2[n2--];

        }else if(n2<0){

            nums1[traverse]=nums1[n1--];

        }else{

            //We can just compare elements in nums1 and nums2

            if(nums1[n1]>nums2[n2]){
                nums1[traverse]=nums1[n1--];
            }else{
                nums1[traverse]=nums2[n2--];
            }
        }

        traverse--;
    }
}
```

Time Complexity is O(m+n) cuz we're iterating through nums1 & nums2
Space Complexity is O(1)

Merge intervals

(Link to the problem: https://leetcode.com/problems/merge-intervals/)

Example:
Input: [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].

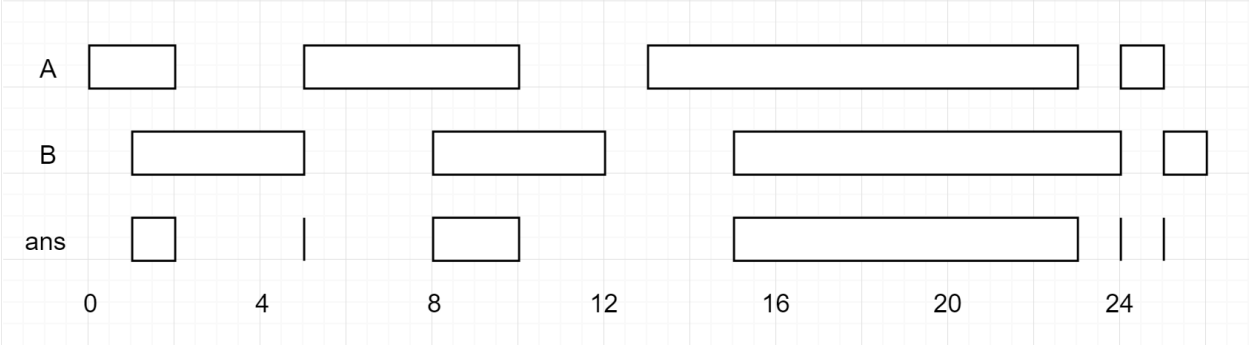
This is a very commonly asked interview question. You can refer to the video to understand the approach for this problem.
https://youtu.be/qKczfGUrFY4
Don't forget to ask your interviewer whether or not the intervals are disjointed. The given solution is only for disjointed intervals.

```
vector<vector<int>> merge(vector<vector<int>>& intervals) {
    vector<vector<int>>ans;
    if(intervals.size()==0)return ans;
   sort(intervals.begin(),intervals.end());

    ans.push_back(intervals[0]);

    for(int i=1;i<intervals.size();i++){

        if(ans.back()[1]>=intervals[i][0]){
            ans.back()[1]=max(ans.back()[1],intervals[i][1]);
        }else ans.push_back(intervals[i]);
    }

    return ans;
}
```

Interval List intersections

(Link to the problem: https://leetcode.com/problems/interval-list-intersections/)

Example:
Input: A = [[0,2],[5,10],[13,23],[24,25]], B = [[1,5],[8,12],[15,24],[25,26]]
Output: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]



Unlike the previous problem, here the list is already given in the ascending order and hence we don't have to sort it.

Assume an interval like this:
start1(s1) -------|------| end1(e1)
start2(s2) |------|------- end2(e2)

In order for an interval to intersect, **s1<=e2** and **s2<=e1**
Now, if they're intersecting, the interval in which they are intersecting is **[max(s1,s2), min(e1,e2)]**

The way we will implement the solution is by having two pointers A and B for traversing through the two lists, and once we find an intersection we will push it in our output vector and then to decide whether to increment A or B we will have to check their second value, i.e. if
A = [0,2] and B = [1,5] the intersection will be [1,2] and now since the span of B(5) is larger than the span of A(2), we will increment A since there might be further intersections possible with B.

```cpp
vector<vector<int>> intervalIntersection(vector<vector<int>>& A, vector<vector<int>>& B) {

    int A_pointer=0, B_pointer=0;

    vector<int> temp(2);
    vector<vector<int>> result;

    while(A_pointer<A.size() && B_pointer<B.size()){

        if(A[A_pointer][0]<=B[B_pointer][1] && A[A_pointer][1]>=B[B_pointer][0]){
            //It intersects

            temp[0] = max(A[A_pointer][0],B[B_pointer][0]);
            temp[1] = min(A[A_pointer][1],B[B_pointer][1]);

            result.push_back(temp);
        }

        if(A[A_pointer][1]>B[B_pointer][1]){

            B_pointer+=1;

        }else{

            A_pointer+=1;

        }
    }

    return result;
}
```

DAY 2 (Arrays)

Set Matrix Zeroes.

(Link to the problem: https://leetcode.com/problems/set-matrix-zeroes/)

Example:
Input:
[
  [0,1,2,0],
  [3,4,5,2],
  [1,3,1,5]
]
Output:
[
  [0,0,0,0],
  [0,4,5,0],
  [0,3,1,0]
]

This problem isn't very hard to solve but since there is a constraint that we must solve it in constant space or in O(1) space, it gets a little tricky.

You can refer to this video for a really good explanation of the problem https://www.youtube.com/watch?v=6_KMkeh5kEc

```
void setZeroes(vector<vector<int>>& matrix) {
    /*
    Instead of storing a separate array for column status
    We store it in the 1st row, and replace those columns that have 0s in subsequent rows
    and keep the rest the same. Now changing this might affect how we check the subsequent
    rows as the first row might not neccessarily have a 0 initially, so we need to check
    at the start whether or not first row should be 0.
    */
    int H = matrix.size();
    int W = matrix[0].size();

    bool isFirstRowZero=false;

    for(int col=0; col<W; ++col){
        if(matrix[0][col]==0){
            isFirstRowZero=true;
        }
    }

    /*
    Now let's modify the first row based on the subsequent rows
    */
    for(int row=1; row<H; ++row){
        for(int col=0; col<W; ++col){
            if(matrix[row][col]==0){
                matrix[0][col]=0;
            }
        }
    }
```

```
    /*
    Now we will first check whether or not a row should be zero, if it should be then
    the entire row becomes 0, or if that column in the first row is 0, then that element
    becomes 0
    */

    for(int row=1; row<H; ++row){
        bool isRowZero=false;
        for(int col=0; col<W; ++col){
            if(matrix[row][col]==0){
                isRowZero=true;
                break;
            }
        }

        for(int col=0; col<W; ++col){
            if(isRowZero || matrix[0][col]==0){
                matrix[row][col]=0;
            }
        }
    }

    /*
    We can now fill the first row with zeros if required
    */

    if(isFirstRowZero){
        for(int col=0; col<W; ++col){
            matrix[0][col]=0;
        }
    }

}
```

Pascals Triangle

(Link to the problem: https://leetcode.com/problems/pascals-triangle/)

This problem is fairly simple and straightforward. We are just using the previous row's values and calculating new values.
Every first and last element of a row is 1.

```
vector<vector<int>> generate(int numRows) {
    vector<vector<int>> result;
    if(numRows==0) return result;
    vector<int> temp{1};
    result.push_back(temp);
    for(int row=1; row<numRows; ++row){
        vector<int> inter{1};
        for(int i=1; i<row; ++i){
            inter.push_back(result[row-1][i-1]+result[row-1][i]);
        }
        inter.push_back(1);
        result.push_back(inter);
    }
    return result;
}
```

Count number of inversions

(Link to the problem: https://www.interviewbit.com/problems/inversion-count-in-an-array/)

Example:
Input: [1, 2, 5, 4, 3]
Output: 3

Approach 1:

The easy approach for this problem would be to go through the array for every element at index i from i+1 to n-1, where n is the size of the array, and checking if any element is larger than the current one, if it is then we will increase the inversion count.

Time Complexity is $O(n^2)$

Approach 2:

For this approach, we will make use of the merge sort algorithm.
You can refer to this video to understand the working of merge sort:
https://www.youtube.com/watch?v=yzkzQ7oZwIE
To solve this problem will make use of the step when we merge the two sorted arrays. Assume the two arrays are [4,9,10] and [2,3] and a pointer i that is pointing towards the start of the 1st array. Now while comparing the elements, when we compare 4 and 2 we see that 4>2 and since the first array is sorted, every element after 4 is also greater than 2, and hence the total number of inversions relating to 2 is **(sizeOfArray1-i)**. Similarly, we will go on and then compare 4 with 3 and again the total number of inversions relating to 3 is 3(3-0).

```
int merge(int a[],int s,int e){
    int mid = (s+e)/2;
    int inversions=0;

    int i = s;
    int j = mid+1;
    int k = s;

    int temp[100];

    while(i<=mid && j<=e){
        if(a[i] < a[j]){
            temp[k++] = a[i++];
        }
        else{
            inversions+=mid-i+1;
            temp[k++] = a[j++];
        }
    }
    while(i<=mid){
        temp[k++] = a[i++];
    }
    while(j<=e){
        temp[k++]  = a[j++];
    }

    for(int i=s;i<=e;i++){
        a[i] = temp[i];
    }

    return inversions;

}
```

```
int mergeSort(int a[],int s,int e){
    int inversions=0;

    if(s>=e){
        return inversions;
    }

    int mid = (s+e)/2;

    inversions+=mergeSort(a,s,mid);
    inversions+=mergeSort(a,mid+1,e);

    inversions+=merge(a,s,e);
    return inversions;
}
```

Time Complexity is O(N log N)

Buy and sell stock

(Link to the problem: https://leetcode.com/problems/best-time-to-buy-and-sell-stock/)

Example:
Input: [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5. Not 7-1 = 6, as the selling price needs to be larger than buying price.

The answer for this problem is just a simple greedy approach. We need to find two values that come one after another such that the former is greater than the later and the difference between them is maximum.

```
int maxProfit(vector<int>& prices) {
    int min_price = INT_MAX;
    int profit = 0;
    for(int i=0; i<prices.size(); ++i){
        if(prices[i]<min_price){
            min_price = prices[i];
        }else if(prices[i]-min_price>profit){
            profit = prices[i]-min_price;
        }
    }

    return profit;
}
```

Time Complexity is O(N)

Rotate a matrix

(Link to the problem: https://leetcode.com/problems/rotate-image/)

Example:
Given **input matrix** =
[
  [5, 1, 9,11],
  [2, 4, 8,10],
  [13, 3, 6, 7],
  [15,14,12,16]
],

rotate the input matrix **in-place** such that it becomes:
[
  [15,13, 2, 5],
  [14, 3, 4, 1],
  [12, 6, 8, 9],
  [16, 7,10,11]
]

It might not strike you at the start but if you recall matrix operations, then you might remember something called transpose of a matrix. We will do a similar operation to get our desired result.
First lets see what happens if we take the transpose of the given matrix.
[
  [5, 2, 13,15],
  [1, 4, 3,14],
  [9, 8, 6, 12],
  [11,10,7,16]
]

Now, if you look closely and observe the above matrix, you'll realise that if we take the mirror image of the above matrix about the y-axis, we will get a matrix rotated by 90deg, and hence the required answer.

```
void rotate(vector<vector<int>>& matrix) {

    int H = matrix.size();
    int W = matrix[0].size();

    //Taking the transpose
    for(int row=0; row<H; ++row){
        for(int col=0; col<(W-row); ++col){
            swap(matrix[row][col+row],matrix[col+row][row]);
        }
    }

    //Finding out the mirror image
    for(int row=0;row<H;++row){
        for(int col=0;col<(W/2);++col){
            swap(matrix[row][col],matrix[row][W-col-1]);
        }
    }
}
```

Time Complexity is $O(n^2)$

Next Permutation

(Link to the problem: https://leetcode.com/problems/next-permutation/)

Example:
1,2,3 → 1,3,2

3,2,1 → 1,2,3

1,1,5 → 1,5,1

We basically have to find the **next greatest** permutation, or permutation that is just greater than the given number.

Let's take a few examples to understand the problem further. Assume nums=[5,4,3,2,1], now if you notice, this is the largest possible permutation, so we can't have any other permutation greater than this and we will just return the smallest possible permutation [1,2,3,4,5].

Now, let's take another example where nums=[1,2,7,9,8,6,4]. If we look at this from the right side, you will notice it is ascending from right to left till we reach 7(index=2). This means that we can't make any swap from index 3 till index 6 because that would always result in a smaller number. But what if we find the next number greater than 7 in [3,6] and swap that with 7. In this case, 8 is the next greatest and if we swap our nums would look like this; nums=[1,2,8,9,7,6,4]. Now if you notice, elements in [3,6] are still in descending order, and if we reverse them we will get the smallest possible number with those numbers and then will get the next greatest permutation, nums=[1,2,8,4,6,7,9].
If we look at smaller examples with length 2, then it can either be the greatest or the smallest, so we can just reverse it and return.

```cpp
void reverse(vector<int>&nums, int start, int end){
    while(start<end){
        swap(nums[start++],nums[end--]);
    }
}
void nextPermutation(vector<int>& nums) {
    if(nums.size()==1) return;
    if(nums.size()==2) return reverse(nums, 0, 1);

    int dec= nums.size()-2;
    while(dec>=0 && nums[dec]>=nums[dec+1]){
        dec--;
    }
    reverse(nums, dec+1, nums.size()-1);
    if(dec==-1){
        return;
    }
    int next_greatest=dec+1;
    while(next_greatest<nums.size() && nums[next_greatest]<=nums[dec]){
        next_greatest++;
    }
    swap(nums[dec],nums[next_greatest]);
}
```

DAY 3 (Math/Brain Teasers)

(pending)Excel Sheet Column Number

(Link to the problem: https://leetcode.com/problems/excel-sheet-column-number/)

Example:
Input: "ZY"
Output: 701

The problem is fairly simple once you figure out the pattern.

```
int titleToNumber(string s) {
    int colNumber = 0;
    for(int i=s.length();i>0;i--){
        colNumber += (s[i - 1] - 'A' + 1) * pow(26, (s.length() - i));
    }
    return colNumber;
}
```

//Will add the rest for this day soon

DAY 4 (Hashing)

Two Sum

(Link to the problem: https://leetcode.com/problems/two-sum/)

Example:
Given nums = [2, 7, 11, 15], target = 9,

Because nums[**0**] + nums[**1**] = 2 + 7 = 9,
return [**0**, **1**].

For this problem we will make use of a hashmap. If the array was sorted, we could've used a two-pointer approach to solve it, but that's for later. The way we can solve this problem using a hashmap is by storing nums[i] and i in the hashmap where i is the index of an element in the nums array. Whenever we encounter a new element, we find its complement i.e. target - nums[i], and we will check if the complement exists in the hashmap or not.

```
vector<int> twoSum(vector<int>& nums, int target) {
    vector<int> numbers;
    if(nums.size()==0) return numbers;
    unordered_map<int,int> umap;
    for(int i=0;i<nums.size();i++){
        int cmpl = target - nums[i];
        if(umap.find(cmpl)!= umap.end()){
            numbers.push_back(umap.find(cmpl)->second);
            numbers.push_back(i);
            break;
        }
        umap.insert(make_pair(nums[i],i));
    }

    return numbers;
}
```

Longest Consecutive Sequence

(Link to the problem: https://leetcode.com/problems/longest-consecutive-sequence/)

Example:
Input: [100, 4, 200, 1, 3, 2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4].

Approach 1:

For the first approach we can sort the array and then keep checking how long a sequence lasts and if that sequence is larger than the current maximum sequence that we have. I've implemented it using a **set**, but you can first sort it and then do similar operations as well. The time complexity of the solution is O(nlogn), but the problem requires us to solve it in O(n) time complexity; approach 2 will explain that solution.

```
int longestConsecutive(vector<int>& nums) {
    set<int> hset;
    if(nums.size()==0) return 0;
    for(int i=0;i<nums.size();i++){
        hset.insert(nums[i]);
    }

    auto it = hset.begin();
    int temp= *it;
    int maxLen=1;
    int tempLen=1;
    while(it!=hset.end()){
        if((*it)-1==temp){
            tempLen++;
            maxLen= max(tempLen,maxLen);
        }else{
            tempLen=1;
        }
        temp=*it;
        it++;
    }
    return maxLen;
}
```

Approach 2:

To solve this problem in O(n) time, we will make use of **unordered_set(in C++)** or **HashSet(in Java)**. These are implemented using a hash table and hence takes constant time for its operations.

The idea for this problem is whenever we encounter a new number, we will start moving in both left and right directions till the numbers are consecutive and keep removing the numbers to avoid repetitions. This will become more clear once you go through the code.

```
int longestConsecutive(vector<int>& nums) {
     unordered_set<int> hset;
     if(nums.size()==0) return 0;
     for(int i=0;i<nums.size();i++){
         hset.insert(nums[i]);
     }

     auto it = hset.begin();
     int temp;
     int maxLen = 1;
     while(!hset.empty()){
         temp = *it;
         hset.erase(temp);
         int lowerBound = temp-1;
         while(hset.find(lowerBound)!=hset.end()){
             hset.erase(lowerBound);
             lowerBound--;
         }
         int upperBound = temp+1;
         while(hset.find(upperBound)!=hset.end()){
             hset.erase(upperBound);
             upperBound++;
         }
         maxLen= max(maxLen,upperBound - lowerBound -1);
         it=hset.begin();
     }
     return maxLen;
 }
```

lowerBound moves towards the left, for example (4->3->2->1) and ***upperBound*** moves towards the right, for example (4->5->6->7).

Largest subarray with 0 sum

(Link to the problem: https://practice.geeksforgeeks.org/problems/largest-subarray-with-0-sum/1)

Example:
Input: [15 -2 2 -8 1 7 10 23]

Output: 5

Explanation: In the above test case the largest subarray with sum 0 will be -2 2 -8 1 7.

The way we'll approach this problem is by calculating the running sum of the array.

Element	2	2	-4	1	-1	0
Running Sum	2	4	0	1	0	0

If you notice, there are three cases that we can see here:

1. **Running Sum is 0:** For this case if you look carefully, whenever the running sum is 0, the maximum length of the subarray with 0 sum is i+1, where i is the index at which the running sum becomes 0.

2	2	-4
2	4	0

2. **Running Sum repeats:** If you notice whenever the running sum repeats, the longest subarray is **i-j,** where j is the index with the

first occurrence and i is the index with the second occurrence.

-4	1	-1	0
0	1	0	0

3. **Max length is 0 and element is 0:** Imagine if you have an array like [0] then max length possible is 1.

```
int maxLen(int A[], int n)
 {
     unordered_map<int,int> runningSum; //<runningSum,index>
     int tempRun = 0;
     int maxLen = 0;
     for(int i=0;i<n;i++){
         tempRun+=A[i];
         if(A[i]==0 && maxLen==0){
             maxLen=1;
         }
         if(tempRun==0){
             maxLen = i+1;
         }

         if(runningSum.find(tempRun)!=runningSum.end()){
             maxLen = max(maxLen,i-(runningSum.find(tempRun)->second));
         }else{
             runningSum.insert(make_pair(tempRun,i));
         }
     }

     return maxLen;
 }
```

XOR Queries of a Subarray

(Link to the problem: https://leetcode.com/problems/xor-queries-of-a-subarray/)

Example:
Input: arr = [1,3,4,8], queries = [[0,1],[1,2],[0,3],[3,3]]
Output: [2,7,14,8]
Explanation:
The binary representation of the elements in the array are:
1 = 0001
3 = 0011
4 = 0100
8 = 1000
The XOR values for queries are:
[0,1] = 1 xor 3 = 2
[1,2] = 3 xor 4 = 7
[0,3] = 1 xor 3 xor 4 xor 8 = 14
[3,3] = 8

Like for the previous problem we created a running sum array, we will create something called the prefix array which will contain the running XORS of all the elements.
Imagine two intervals, [a,b] and [a,c] where c>b. Consider the running XORS of the first interval to be A and for the second interval to be B. If we do the following operation, A^B we get some result C. But we know about the properties of XOR, we know that the two same numbers cancel out each other and become 0, i.e. 3^3=0 and 4^4=0 and so on.. This means that, C is basically the running XOR for the interval [b,c] and using this knowledge we can solve the above problem.

```
 void prefixGeneration(vector<int>& prefix, vector<int>& arr){
    int tempxor = 0;
    for(int i=0;i<arr.size();i++){
        tempxor^=arr[i];
        prefix.push_back(tempxor);
    }
}

vector<int> xorQueries(vector<int>& arr, vector<vector<int>>& queries) {
    vector<int> prefix;
    vector<int> result;
    prefixGeneration(prefix,arr);
    for(int i=0;i<queries.size();i++){
        vector<int> query = queries[i];
        int B = prefix[query[1]];
        if(query[0]-1<0){
            result.push_back(B);
        }else{
            int A = prefix[query[0]-1];
            result.push_back(A^B);
        }
    }
    return result;
}
```

Longest Substring Without Repeating Characters

(Link to the problem:
https://leetcode.com/problems/longest-substring-without-repeating-characters/)

Example:
Input: "pwwkew"
Output: 3
Explanation: 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.

For solving this problem we will make use of the sliding window method, i.e. we keep a range [i,j] between which we will perform our operation.

Now, consider a string “abcabcbb”. The idea is, if there is a substring between [i,j-i] which has no repeating characters, then we just have to check if j^{th} character has already occurred in that [i,j-1] range or not, if it hasn’t then it is also a part of the substring. If the j^{th} character has occurred before, say at j' position, then the j^{th} character can only be part of the string from the $(j+1)^{th}$ position.

For example, for the string “abcabcbb”, assume you current window is [0,2] and s[3] is ‘a’ which has already occurred at 0, hence the new valid substring is in the range [1,3].
Assume another string “abcefcda” and imagine that our current window is [3,6], “abcefcda”, now if we were to add ‘a’ we will check when was the last time that ‘a’ had occurred, we see that it had occurred at 0, but the lower index of the current window is greater than 0, and hence the valid substring will start at 3.

Therefore, **valid_substr_start = max(lower index, prev occurrence)**

```
int lengthOfLongestSubstring(string s) {
    int ans=0;
    unordered_map<char,int> umap;
    for(int i=0,j=0;j<s.length();j++){
        if(umap.find(s[j])!=umap.end()){
            i= max(i,umap[s[j]]);
        }
        ans = max(ans,j-i+1);
        umap[s[j]]=j+1;
    }
    return ans;
}
```

Here, i is the lower index of the current window and the unordered_map is used for storing the last occurrences of the characters.

DAY 5 (LinkedList)

Reverse Linked List

(Link to the problem: https://leetcode.com/problems/reverse-linked-list/)

Example:
Input: 1->2->3->4->5->NULL
Output: 5->4->3->2->1->NULL

Iterative Approach:

For this approach we will make use of 3 pointers: prev, curr and temp.

```
ListNode* reverseList(ListNode* head) {
    ListNode* prev=NULL, curr=head;

    while(curr!=NULL){
        ListNode* tempNode = curr->next;
        curr->next = prev;
        prev= curr;
        curr= tempNode;
    }

    return prev;
}
```

Time Complexity is O(n)
Space Complexity is O(1)

Recursive Approach:

```
ListNode* reverseList(ListNode* head) {
    if(head == NULL || head->next == NULL) return head;

    ListNode* node = reverseList(head->next);

    head->next->next = head;//This is for reversing
    head->next= NULL; //This is for making the original head point towards NULL

    return node;
}
```

Time Complexity is O(n)
Space Complexity is O(n) because of stack space during recursion

Find middle of LinkedList

(Link to the problem: https://leetcode.com/problems/middle-of-the-linked-list/)

Example:
Input: [1,2,3,4,5,6]
Output: Node 4 from this list (Serialization: [4,5,6])
Since the list has two middle nodes with values 3 and 4, we return the second one.

Approach 1:

For this approach we will find the length of the linked list and then traverse again till length/2.

```
ListNode* middleNode(ListNode* head) {
    int length=0;
    ListNode* tempNode = head;
    while(tempNode){
        length++;
        tempNode=tempNode->next;
    }

    length /=2;

    tempNode = head;
    while(length--){
        tempNode = tempNode->next;
    }

    return tempNode;
}
```

Approach 2:

We will make use of fast and slow pointers, similar to tortoise and hare problem on Day 1. This works because fast covers twice the distance that slow covers.

```
ListNode* middleNode(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;
    while(fast != NULL && fast->next!=NULL){
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

Merge two sorted linked lists

(Link to the problem: https://leetcode.com/problems/merge-two-sorted-lists/)

Example:
Input: 1->2->4, 1->3->4
Output: 1->1->2->3->4->4

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    ListNode* new_head = NULL;
    if(l1==NULL) return l2;
    if(l2==NULL) return l1;

    ListNode* sorting_pointer=NULL;
    if(l1&&l2){
        if(l1->val<=l2->val){
            sorting_pointer = l1;
            l1 = sorting_pointer->next;
        }else{
            sorting_pointer = l2;
            l2 = sorting_pointer->next;
        }
    }

    new_head = sorting_pointer;

    while(l1&&l2){
        if(l1->val<=l2->val){
            sorting_pointer->next=l1;
            sorting_pointer = l1;
            l1 = sorting_pointer->next;
        }else{
            sorting_pointer->next=l2;
            sorting_pointer = l2;
            l2 = sorting_pointer->next;
        }
    }

    if(l1==NULL) sorting_pointer->next = l2;
    if(l2==NULL) sorting_pointer->next = l1;

    return new_head;
}
```

We have used the merge concept of merge sort to solve this problem.

Remove Nth Node From End of List

(Link to the problem: https://leetcode.com/problems/remove-nth-node-from-end-of-list/)

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**.

```
ListNode* removeNthFromEnd(ListNode* head, int n) {
    if (!head) return nullptr;

    ListNode new_head(-1);
    new_head.next = head;

    ListNode *slow = &new_head, *fast = &new_head;

    for (int i = 0; i < n; i++)
        fast = fast->next;

    while (fast->next)
    {
        fast = fast->next;
        slow = slow->next;
    }

    slow->next = slow->next->next;

    return new_head.next;
}
```

We are moving the fast pointer n-steps ahead of the slow pointer.

Delete a given Node when a node is given.

(Link to the problem: https://leetcode.com/problems/delete-node-in-a-linked-list/)

Example:
Input: head = [4,5,1,9], node = 5
Output: [4,1,9]
Explanation: You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.

```
void deleteNode(ListNode* node) {
    node->val = node->next->val;
    node->next=node->next->next;
}
```

Add two numbers as LinkedList

(Link to the problem: https://leetcode.com/problems/add-two-numbers/)

Example:
Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8
Explanation: 342 + 465 = 807

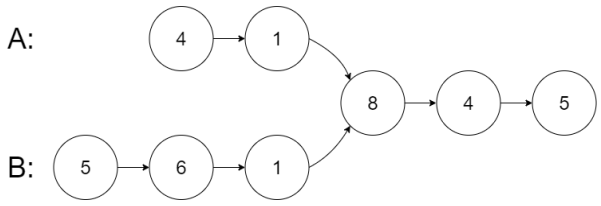
```
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    ListNode dummyHead(-1);
    ListNode *curr = &dummyHead;
    int carry=0;
    while(l1 || l2){
        int x = (l1)?l1->val:0;
        int y = (l2)?l2->val:0;
        int sum=x+y+carry;
        carry = sum/10;
        curr->next = new ListNode(sum%10);
        curr=curr->next;
        if(l1) l1=l1->next;
        if(l2) l2=l2->next;
    }

    if(carry>0){
        curr->next= new ListNode(carry);
    }

    return dummyHead.next;
}
```

DAY 6 (LinkedList)

Intersection of Two Linked Lists

(Link to the problem: https://leetcode.com/problems/intersection-of-two-linked-lists/)

Example:



Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3
Output: Reference of the node with value = 8
Input Explanation: The intersected node's value is 8 (note that this must not be 0 if the two lists intersect). From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8,4,5]. There are 2 nodes before the intersected node in A; There are 3 nodes before the intersected node in B.

Approach 1

For the first approach we will make use of hashing. We will make a hashmap that has the key value as the node and as we keep traversing the lists we will keep checking if the current node exists in the hashmap, if it does, then that is the intersection point.

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    unordered_map<ListNode*,bool> visitedNodes;
    if(!headA || !headB) return NULL;

    while(headA){
        visitedNodes[headA]=true;
        headA=headA->next;
    }

    while(headB){
        if(visitedNodes.find(headB) != visitedNodes.end()){
            return headB;
        }
        visitedNodes[headB]=true;
        headB=headB->next;
    }

    return NULL;
}
```

Time Complexity is O(m+n)
Space Complexity is O(m) or O(n)

But as you notice the space complexity isn't O(1). So let's see if we can improve this.

Approach 2

First I'll explain the approach of the problem and then I'll explain why it works. Consider two lists A and B, the length of A is 5 and that of B is 3. Now initialize two pointers to the start of each list and keep traversing.

You'll notice the B pointer will reach the end of the list first, now initialize this pointer to the start of list A, similarly when A pointer reaches the end initialize that to the start of list B.

Why did we do that? You will notice, the length of A is 2 more than B, so by reinitializing pointer B to the start of A, we make B move two steps

ahead and so by the time pointer A gets initialized to the start of list B, both pointer A and pointer B will have to cover the same distance to reach the end of the lists and hence they are equidistant from the intersection point.

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    ListNode* ptA= headA;
    ListNode* ptB= headB;
    if(!ptA || !ptB) return NULL;
    while(ptA && ptB && ptA!=ptB){
        ptA=ptA->next;
        ptB=ptB->next;
        if(ptA==ptB) return ptA;
        if(!ptA) ptA=headB;
        if(!ptB) ptB=headA;
    }
    return ptA;
}
```

Detect and remove loop

(Link to the problem: https://www.interviewbit.com/problems/detect-and-remove-loop-from-a-linked-list/)

Example:
Input:

```
3 -> 2 -> 4 -> 5 -> 6
       ^       |
       |       |
       - - - - - -
```

Output: 3 -> 2 -> 4 -> 5 -> 6 -> NULL

If you've solved all the previous problems in this series, you should've figured out how to solve this problem now. We will make use of the same tortoise and hare algorithm that we had used in 'Find the duplicate number' problem on day 1.

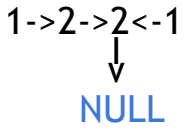
```
ListNode* Solution::solve(ListNode* A) {
    ListNode* slow = A;
    ListNode* fast = A;
    while(1){
        slow = slow->next;
        fast = fast->next->next;
        if(slow==fast) break;
    }
    fast=A;
    while(slow!=fast){
        slow=slow->next;
        fast=fast->next;
    }
    while(fast->next != slow){
        fast=fast->next;
    }

    fast->next=NULL;

    return A;
}
```

Palindrome Linked List

(Link to the problem: https://leetcode.com/problems/palindrome-linked-list/)

Example:
Input: 1->2->2->1
Output: true

Our goal is to solve this problem in constant space. What we will first do is find the middle of the linked list. Then, we will reverse the linked list from there. So if our linked list initially is 1->2->2->1 the middle would be 1->2->2->1. Now, when we reverse the linked list from there onwards our resultant linked list looks like this:



We can now just travel the linked list from both the ends with two pointers and keep checking the values to verify whether it's a palindrome or not.

```
bool isPalindrome(ListNode* head) {
    ListNode* slow=head;
    ListNode* fast=head;
    ListNode* prev=NULL;
    while(fast && fast->next){
        slow=slow->next;
        fast=fast->next->next;
    }
    while(slow){
        ListNode* nextNode = slow->next;
        slow->next=prev;
        prev=slow;
        slow=nextNode;
    }
    fast=head;
    while(prev!=NULL){
        if(prev->val!=fast->val){
            return false;
        }
        prev=prev->next;
        fast=fast->next;
    }
    return true;
}
```

Reverse Nodes in K-groups

(Link to the problem: https://leetcode.com/problems/reverse-nodes-in-k-group/)

Example:
Given this linked list: 1->2->3->4->5

For *k* = 2, you should return: 2->1->4->3->5

For *k* = 3, you should return: 3->2->1->4->5

I'll explain my step by step approach to this problem.

1. For the example we see that, a group should be rotated only if it's a group of size exactly = k. So this way I realised that I'll have to calculate the valid number of groups. To do this I just calculated the length of the linked list and divided it by k, that would give us the valid number of groups(=G).
2. Now I run a while loop G number of times because I need to rotate the only G groups and rest remain the same.
3. Consider this example when k=2 and the list is like this 1->2->3->4->5. If we apply the reversing a linked list logic to reverse the first group or the first k-node after the first group our linked list would like this: 1<-2 3->4->5, so our goal is to reverse the group and link the original head of the previous group(in this case 1) to the new head of the next group(in this case 4). To do this, we just need to add a new pointer that keeps track of the original head of some group and because we know from our previous knowledge that after we rotate some part of the linked list, we always get a pointer to the new head of the reversed group. In our code, the variable 'start' keeps the track of the original head.

```
ListNode* reverseKGroup(ListNode* head, int k) {
    ListNode* prev = NULL, *start=NULL,
    *returnNode=NULL,*lengthNode=head;
    int x=k,iter=0,length=0;

    while(lengthNode){
        length++;
        lengthNode=lengthNode->next;
    }
    int groups = length/k;

    while(groups--){
        ListNode* curr=head,*tempHead=head,*nextNode=NULL;
        while(k--){
            nextNode= curr->next;
            curr->next=prev;
            prev=curr;
            curr=nextNode;
            if(iter==0) iter=1;
        }
        if(iter==1){
            returnNode=prev;
            iter++;
        }
        k=x;
        if(start) start->next=prev;
        if(!nextNode) break;
        start=tempHead;
        head=nextNode;
        if(groups==0) start->next=head;
        prev=NULL;
    }
    return returnNode;
}
```

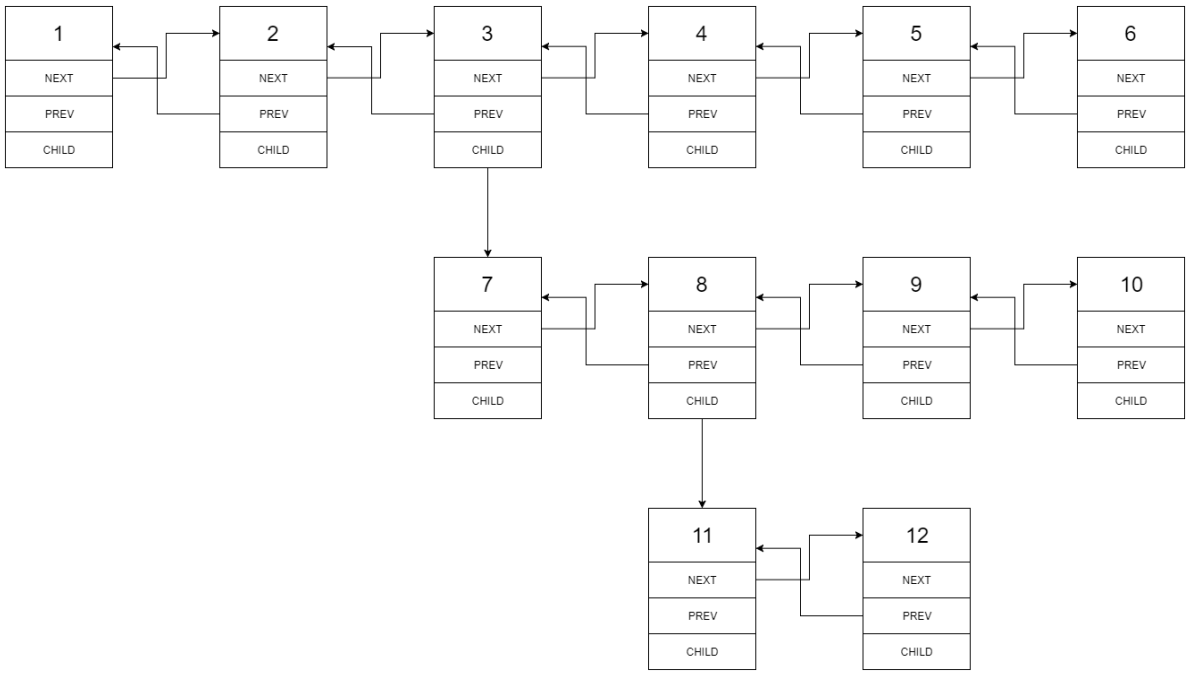
Flatten a multilevel doubly linked list

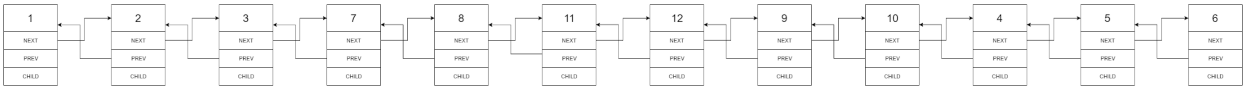
(Link to the problem: https://leetcode.com/problems/flatten-a-multilevel-doubly-linked-list/)

Example:
Input: head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]

Output: [1,2,3,7,8,11,12,9,10,4,5,6]

Explanation: The multilevel linked list in the input is as follows:





The first thing that came to my mind was to somehow store the pending processes whenever a child node is encountered. To do so I have used a stack, this will allow us to get the latest pending processes.

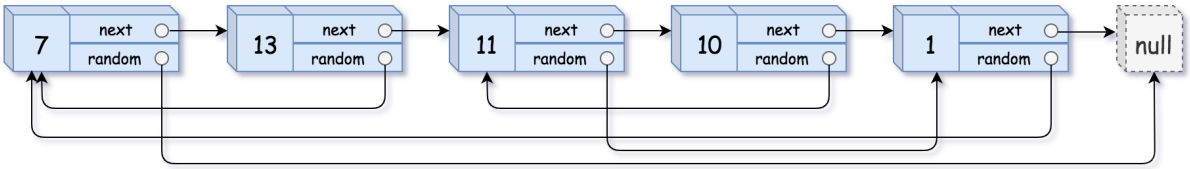
```
Node* flatten(Node* head) {
    stack<Node*> pendingNodes;
    Node* currNode= head;
    while(currNode){
        if(currNode->child){
            if(currNode->next){
                pendingNodes.push(currNode->next);
            }
            currNode->child->prev=currNode;
            currNode->next=currNode->child;
        }
        if(!(currNode->next) && !pendingNodes.empty()){
            currNode->next=pendingNodes.top();
            currNode->next->prev=currNode;
            pendingNodes.pop();
        }
        currNode->child=NULL;
        currNode=currNode->next;

    }
    return head;
}
```

Copy List with Random Pointer

(Link to the problem: https://leetcode.com/problems/copy-list-with-random-pointer)

Example:
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

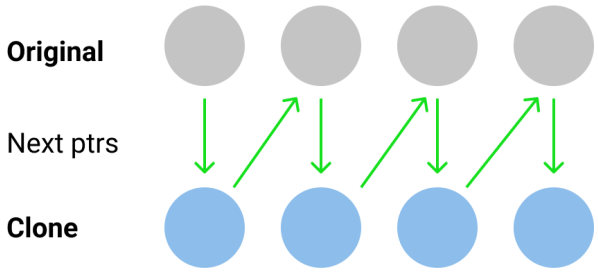
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]



This is a tricky problem and to understand it properly you'll need to use diagrams, it'll be easier to understand if you watch this short video. https://youtu.be/EHpS2TBfWQg

The explanation of the code is as follows:

1. First we have just cloned the values and the nodes. We have also made all the next pointers of the original list point to the respective new cloned node



2. Next we have copied the random pointers with the help of the above implementation.

3. After this we have just restored the original list and linked the new nodes with each other.

```
Node* copyRandomList(Node* head) {
    if(!head) return head;
    Node* clone_head=nullptr, *curr_node=head, *curr_clone=nullptr;

    //In this step we have just cloned the values and the nodes. We have also
    //made all the next pointers of the original list point to the respective new cloned node

    while(curr_node){
        curr_clone = new Node(curr_node->val, curr_node->next, nullptr);
        curr_node->next = curr_clone;
        curr_node = curr_clone->next;
    }

    curr_node=head;

    //Here we have assigned the random pointers
    while(curr_node){
        curr_clone=curr_node->next;
        if(curr_node->random) curr_clone->random = curr_node->random->next;
        curr_node = curr_clone->next;
    }

    curr_node=head;
    clone_head= head->next;
    while(curr_node){
        curr_clone=curr_node->next;
        curr_node->next = curr_clone->next;
        curr_node=curr_node->next;
        if(curr_clone->next) curr_clone->next= curr_clone->next->next;
    }

    return clone_head;

}
```

Rotate List

(Link to the problem: https://leetcode.com/problems/rotate-list/)

Example:
Input: 0->1->2->NULL, k = 4
Output: 2->0->1->NULL
Explanation:
rotate 1 steps to the right: 2->0->1->NULL
rotate 2 steps to the right: 1->2->0->NULL
rotate 3 steps to the right: 0->1->2->NULL
rotate 4 steps to the right: 2->0->1->NULL

The answer for this question is pretty straightforward, the only thing you need to take care of is the situation when the number of rotations is greater than that of the length of the entire list.

```
ListNode* rotateRight(ListNode* head, int k) {
    if(!head || k==0) return head;
    ListNode* traverse=head, *prevEnd=NULL;
    int length=0;

    while(traverse){
        length++;
        prevEnd=traverse;
        traverse=traverse->next;
    }

    if(k%length==0) return head;
    int distance= length-(k%length)-1;

    ListNode* newEnd=head;
    while(distance--){
        newEnd=newEnd->next;
    }
    ListNode* newHead = newEnd->next;
    newEnd->next=NULL;
    prevEnd->next=head;

    return newHead;
}
```

DAY 7 (Two pointer)

3 sum

(Link to the problem: https://leetcode.com/problems/3sum/)

Example:
Given array nums = [-1, 0, 1, 2, -1, -4],
A solution set is:
[
  [-1, 0, 1],
  [-1, -1, 2]
]
If you've previously solved the 2 sum problem, this wouldn't be difficult to understand. We just have to reduce the 3 sum problem into 2 sum. Instead of finding a+b+c=0, we just need to find b+c=-a for every a in the list. The only thing we need to take care of is handling the duplicates. Once you go through the code, it should become clear.

```
vector<vector<int>> threeSum(vector<int>& nums) {
    vector<vector<int>> solution;
    if(nums.size()<3) return solution;
    sort(nums.begin(),nums.end());

    for(int i=0;i<nums.size()-2;i++){
        if(i==0 || (i>0 && nums[i]!=nums[i-1])){
            int low= i+1;
            int hi=nums.size()-1;
            int target = 0 - nums[i];
            while(low<hi){
                if(nums[low]+nums[hi]==target){
                    vector<int> temp;
                    temp.push_back(nums[i]);
                    temp.push_back(nums[low]);
                    temp.push_back(nums[hi]);
                    solution.push_back(temp);
                    while((low<hi)&&nums[low]==nums[low+1]) low++;
                    while((low<hi)&&nums[hi]==nums[hi-1]) hi--;
                    low++;
                    hi--;
                }else if(nums[low]+nums[hi]>target){
                    hi--;
                }else{
                    low++;
                }
            }
        }
    }

    return solution;
}
```

Max Consecutive Ones

(Link to the problem: https://leetcode.com/problems/max-consecutive-ones/)

Example:
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.

```
int findMaxConsecutiveOnes(vector<int>& nums) {
    int first=0,last=0;
    int max_ones=0;
    for(last;last<nums.size();last++){
        if(last==0 && nums[last]==1 || nums[last]==1 && nums[last-1]==0){
            first=last;
        }
        if(nums[last]==1){
            max_ones = max(max_ones,last-first+1);
        }
    }

    return max_ones;
}
```

Remove Duplicates from Sorted Array

(Link to the problem: https://leetcode.com/problems/remove-duplicates-from-sorted-array/)

Example:
Given *nums* = **[0,0,1,1,1,2,2,3,3,4]**,
Your function should return length = **5**, with the first five elements of *nums* being modified to **0**, **1**, **2**, **3**, and **4** respectively.
It doesn't matter what values are set beyond the returned length.

```
int removeDuplicates(vector<int>& nums) {
    if(nums.size()==0) return 0;
    int i=0;
    for(int j=1;j<nums.size();j++){
        if(nums[i]!=nums[j]){
            i++;
            nums[i]=nums[j];
        }
    }
    return i+1;
}
```

Trapping Rain Water

(Link to the problem: https://leetcode.com/problems/trapping-rain-water/)

Example:

Input: [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6

Watch the following videos to completely understand how to solve such problems. This guy has done an amazing job at explaining the approach.
Part 1: https://www.youtube.com/watch?v=HmBbcDiJapY
Part 2: https://www.youtube.com/watch?v=VZpJxINSvfs
Part 3: https://youtu.be/XqTBrQYYUcc
The key thing to remember is, lmax is always non decreasing and rmax is always non increasing.

```
int trap(vector<int>& height) {
    if(height.size()<3) return 0;

    int total_water=0;
    int i=0, j=height.size()-1, lmax=height[0], rmax= height[height.size()-1];

    while(i<=j){
        lmax= max(lmax,height[i]);
        rmax= max(rmax,height[j]);
        if(lmax<=rmax){
            total_water+= lmax-height[i];
            i++;
        }else{
            total_water+= rmax-height[j];
            j--;
        }
    }

    return total_water;
}
```

DAY 8 (Greedy)

Meeting Rooms

(Link to the problem: https://www.interviewbit.com/problems/meeting-rooms/)

Example:
Input: A = [[1, 18],[18, 23],[15, 29],[4, 15],[2, 11],[5, 13]]
Output: 4
Explanation: Meeting one can be done in conference room 1 from 1 - 18.
Meeting five can be done in conference room 2 from 2 - 11.
Meeting four can be done in conference room 3 from 4 - 15.
Meeting six can be done in conference room 4 from 5 - 13.
Meeting three can be done in conference room 2 from 15 - 29 as it is free in this interval.
Meeting two can be done in conference room 4 from 18 - 23 as it is free in this interval. The reason why we use a minHeap is we essentially just need to find the overlaps. If you remember the merge interval questions, we first sorted the array based on starting time and then checked for overlaps, we are doing something similar here.

```cpp
int Solution::solve(vector<vector<int> > &A) {
    if(A.size()==0) return 0;
    sort(A.begin(),A.end());

    priority_queue<int,vector<int>,greater<int>> minHeap;

    minHeap.push(A[0][1]);

    for(int i=1;i<A.size();i++){
        if(A[i][0]>=minHeap.top()){
            minHeap.pop();
        }

        minHeap.push(A[i][1]);
    }

    return minHeap.size();
}
```

Fractional Knapsack

(Link to the problem: https://www.geeksforgeeks.org/fractional-knapsack-problem/)

Example:
Input: Items as (value, weight) pairs
arr[] = {{60, 10}, {100, 20}, {120, 30}}
Knapsack Capacity, W = 50;
Output: Maximum possible value = 240
by taking items of weight 10 and 20 kg and 2/3 fraction
of 30 kg. Hence total price will be 60+100+(2/3)(120) = 240

To understand how to solve this question properly watch this video
https://youtu.be/oTTzNMHM05I

DAY 9 (Binary Search)

1/N-th root of an integer

(Link to the problem: https://leetcode.com/problems/sqrtx/)

Example:
Input: 4
Output: 2

```
int mySqrt(int x) {
    int left = 1, right = x;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (mid == x / mid) {
        return mid;
        } else if (mid < x / mid) {
        left = mid + 1;
        } else {
        right = mid - 1;
        }
    }
    return right;
}
```

First and Last Occurrence of an element

(Link to the problem:
https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/)

Example:
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]

```
vector<int> searchRange(vector<int>& nums, int target) {
    vector<int> result;
    int fo=-1,lo=-1,start=0,end=nums.size()-1;
    while(start<=end){
        int mid= start + (end-start)/2;
        if(nums[mid]==target){
            fo=mid;
            end=mid-1;
        }else if(nums[mid]>target){
            end=mid-1;
        }else{
            start=mid+1;
        }
    }
    start=0;
    end=nums.size()-1;
    while(start<=end){
        int mid= start + (end-start)/2;
        if(nums[mid]==target){
            lo=mid;
            start=mid+1;
        }else if(nums[mid]>target){
            end=mid-1;
        }else{
            start=mid+1;
        }
    }
    result.push_back(fo);
    result.push_back(lo);
    return result;
}
```

This same concept can be used for finding the total number of occurrences of an element in a sorted array.

total_occurrences = lo-fo+1

Rotation count in a sorted array

(Link to the problem: https://www.geeksforgeeks.org/find-rotation-count-rotated-sorted-array/)

Assume an array nums=[2,5,6,8,11,12,15,18]. If we rotate this array let's say 4 times, the resultant nums=[11,12,15,18,2,5,6,8]. Now, consider this is the input array and our goal is to find the number of times this array is rotated(4).
If we look at the indexes of both the original and rotated array, we notice that when the array is 0 rotated, the minimum element is at index 0 and similarly when array is 4 rotated, the minimum element is at index 4.
So our first step would be to find the minimum element using binary search(since the array is sorted).
In binary search we assume that our target element lies at mid, so let's say the minimum element lies at ***i*** and mid is also equal to ***i***, to check whether this element is the minimum element, nums[i] must satisfy the condition: nums[i-1]>nums[i] and nums[i+1]>nums[i].
Now that we have figured out how to end the search, let's figure out how to reduce our search space.
Say the mid falls at 15, [11,12,15,18,2,5,6,8], we notice that the required element falls in the unsorted part of the array(marked in green). To find the sorted side in this array, we can simply compare a[mid] with a[end], if it is greater, then the unsorted array lies in the right side. Let's look at another example, say nums is [18,2,5,6,8,11,12,15] and mid falls on 6, [18,2,5,6,8,11,12,15], then our unsorted array lies in left side and we can find this our by comparing a[mid] with a[start] and hence reduce the search space accordingly.
To find the previous and next element of the current element we can do the following:

previous = (mid+N-1)%length

next= (mid+1)%length

```
int countRotations(vector<int>& nums) {
    //No rotation
    if(nums[0]<nums[nums.size()-1]) return 0;

    int start=0, end=nums.size()-1, pivot=-1;

    while(start<=end){
        //There might be case when only sorted elements are left
        if(nums[start]<nums[end]) return start;

        int mid=start + (end-start)/2;
        int next = (mid+1)%nums.size();
        int prev = (mid-1+nums.size())%nums.size();
        if(a[mid]<a[next]&&a[mid]<a[prev]){
            pivot=mid;
            break;
        }

        if(a[mid]<=a[end]){
            end=mid-1;
        }else if(a[start]<=a[mid]){
            start=mid+1;
        }
    }

    return pivot;
}
```

Find an element in a Rotated Sorted Array

(Link to the problem: https://leetcode.com/problems/search-in-rotated-sorted-array/)

Example:
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4

If you've understood the previous problem, this would be very easy to understand. Our goal would again be to find the index of the minimum element or the pivot, and then we are left with two different sorted arrays in which we can again apply binary search to find the required number.

Find an element in a Nearly Sorted Array

A nearly sorted array, in this case, is an array where the element a[i] in the given array can lie either at a[i-1], a[i+1] or at a[i] in the actual sorted array.
Example: a=[5,10,30,20,40]
So, how do we search in such an array? Since this array is nearly sorted and not fully sorted, we can't apply the regular binary search, we will have to modify the search a little.
Let's start the approach with the regular binary search , so we first calculate ***mid***, in a fully sorted array we assume the target element exists at a[mid], but now it can exist at a[mid-1], a[mid] or a[mid+1].
Now, if we don't find the number, we reduced the search size by either going to the left or right. Similarly if we compare our target element with a[mid], if a[mid]>target, then instead of ***end=mid-1***, we will do ***end=mid-2***, because we have already checked at ***mid-1***.
While comparing the target element, make sure you check whether ***mid-1*** is less than start(0) and whether ***mid+1*** is greater than end(size-1).

Find the floor and ceiling of a number in a sorted array.

Example: arr=[1,2,3,4,8,9,10] num=5
If 5 exists in arr, then we have to return 5, but if it doesn't let's understand what is the meaning of floor and ceil.By floor we mean that we have to find the number that is at most 5, and by ceil we mean that we have to find the number that is at least 5.In this case the floor of 5 is 4, and the ceiling of 5 is 8.

```
vector<int> floorAndCeil(vector<int>& nums, int target) {
    int start=0, end=nums.size()-1, floor=-1;
    while(start<=end){
        int mid= start + (end-start)/2;
        if(nums[mid]==target){
            floor=target;
            break;
        }
        else if(nums[mid]<target){
            floor = nums[mid]; //This is one possible solution
            start = mid+1;
        }
        else{
            end = mid-1;
        }
    }
    start=0; end=nums.size()-1;
    int ceil=-1;
    while(start<=end){
        int mid= start + (end-start)/2;
        if(nums[mid]==target){
            ceil=target;
            break;
        }
        else if(nums[mid]>target){
            ceil = nums[mid]; //This is one possible solution
            end = mid-1;
        }
        else{
            start = mid+1;
        }
    }
    vector<int> output;
    output.push_back(floor);output.push_back(ceil);
    return output;
}
```

DAY 10 (Binary Search)

Next alphabetical element

(Link to the problem: https://leetcode.com/problems/find-smallest-letter-greater-than-target/)

This question can be solved using the same logic that we have used to find the ceiling of a number.

```
char nextGreatestLetter(vector<char>& letters, char target) {
    int start=0, end=letters.size()-1;
    char next_alphabet;
    while(start<=end){
        int mid=start + (end-start)/2;
        if(letters[mid]==target || letters[mid]<target){
            start=mid+1;
        }else{
            end=mid-1;
        }
    }
    return letters[start%letters.size()];
}
```

Peak Element

(Link to the problem: https://leetcode.com/problems/find-peak-element/)

Example:
Input: nums = [1,2,1,3,5,6,4]
Output: 1 or 5
Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

Now here we apply the same concept of binary search, the only thing that we need to figure out is a way to eliminate the search space. First let's see what's the condition for finding a peak element. For 1<=i<=n-2, a[i] is a peak element if a[i]>a[i-1] and a[i]>a[i+1]. a[0] and a[n-1] are peak elements if a[0]>a[1] and a[n-1]>a[n-2].
Now that we have figured out the terminating condition, let's try and understand how to reduce the search space. Assume an array arr=[1,2,3,4,5], in this array the peak element is '5', here a[mid] in the first iteration is '3' and we have removed the left side of '3'. Now consider another array arr=[4,5,3,2,3], here the peak element is '5', and in the first iteration a[mid] is '3' and we have removed the right side of '3'. So, in order to reach a solution we have to move towards the side which is larger, i.e. for a[mid] we have to move towards left if a[mid-1]>a[mid] or move towards right if a[mid+1]>a[mid]. This will always result in the right answer. Now why does this work? Let's say a[mid+1]>a[mid], now for a[mid+1] to be a peak element, a[mid+2]<a[mid+1], but even if that's not the case and a[mid+2]>a[mid+1] then a[mid+3] has to be smaller else this same pattern keeps going, in this way we will reach the end of the array and since between a[mid] and a[n-1] the numbers are just increasing then a[n-1] is the peak element. This means we are always bound to find a peak element in the side which has a number greater than a[mid].

```
int findPeakElement(vector<int>& nums) {
    if(nums.size()==1) return 0;
    int start=0,end=nums.size()-1;

    while(start<=end){
        int mid= start + (end-start)/2;
        if(mid>0 && mid<nums.size()-1){
            if(nums[mid]>nums[mid+1] && nums[mid]>nums[mid-1]){
                return mid;
            }
            else if(nums[mid+1]>nums[mid]){
                start=mid+1;
            }else if(nums[mid-1]>nums[mid]){
                end=mid-1;
            }
        }else{
            if(mid==0){
                if(nums[mid]>nums[mid+1]){
                    return 0;
                }else{
                    return 1;
                }
            }else if(mid==nums.size()-1){
                if(nums[mid]>nums[mid-1]){
                    return mid;
                }else{
                    return mid-1;
                }
            }
        }
    }

    return 0;
}
```

Search in a 2D row and column sorted Matrix

(Link to the problem: https://leetcode.com/problems/search-a-2d-matrix-ii/)

Example:

```
[
  [10,   20,  30, 40],
  [15,   25,  35, 42],
  [27,   29,  37, 45],
  [32, 33, 39, 56]
]
```

Target is to find '29'

The solution for this problem is pretty interesting. Assume the matrix is a mxm matrix. Take two pointers i=0 and j=m-1, so we'll start with '40'. Now, since the array is sorted we can use this piece of information to go in two directions, either left or bottom. We compare 40 and 29, since 29<40, we can eliminate that column completely and reduce j--. Similarly we compare 29 and 30, and can eliminate that column as well. Now we reach 20. We compare 29 with 20, and since 29>20 with just move one row down or just do i++. Similarly we just keep increasing i and reach 29 eventually. If the number is not present, either i or j will go out of bounds.

```
[
  [10,   20,  30, 40],
  [15,   25,  35, 42],
  [27,   29,  37, 45],
  [32, 33, 39, 56]
]
```

The above is what the path to the target element would look like. All the red colored cells are the eliminated cells.

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    if(matrix.size()==0) return false;
    int i=0,j=matrix[0].size()-1;
    while(i>=0 && i<=matrix.size()-1 && j>=0 && j<=matrix[0].size()-1){
        if(matrix[i][j]==target){
            return true;
        }else if(matrix[i][j]>target){
            j--;
        }else if(target>matrix[i][j]){
            i++;
        }
    }
    return false;
}
```

Time Complexity is O(m+n)

Allocate minimum number of pages(Problem type)

(Link to the problem: https://leetcode.com/problems/capacity-to-ship-packages-within-d-days/)

This is a very important type of binary search based problem. Watch this video to understand it completely
https://www.youtube.com/watch?v=Ss9ta1zmiZo
In these kinds of problems we need to minimize something maximum. Like in the video link, we have minimized the maximum number of pages a student has to read.
In the problem that I have provided in the link, there we have minimize the max capacity a ship has to carry within D days.

```
int isValid(vector<int>& weights, int D, int capacity){
    int dayCount=1;
    int currCapacity=0;
    for(int i:weights){
        currCapacity+=i;
        if(currCapacity>capacity){
            dayCount++;
            currCapacity=i;
        }
    }
    if(dayCount<=D){
        return true;
    }else{
        return false;
    }
}

int shipWithinDays(vector<int>& weights, int D) {
    int start = *max_element(weights.begin(),weights.end());int end =0;
    for(int i:weights){
        end+=i;
    }
    int result = INT_MAX;

    while(start<=end){
        int mid= start + (end-start)/2;
        bool validity = isValid(weights,D,mid);
        if(validity){
            result = mid;
            end=mid-1;
        }
        else{
            start=mid+1;
        }
    }
    return result;
}
```

DAY 11 (Stacks and Queues)

DFS and BFS implementation

Take time to understand how BFS and DFS work and how to implement them using a stack and a queue. Watch this video and understand the working and implementation.
https://youtu.be/uWL6FJhq5fM

Valid Parentheses

(Link to the problem: https://leetcode.com/problems/valid-parentheses/)

Example:
Input: "()[]{}"
Output: true

```
bool isValid(string s) {
    if(s.length()==0) return true;
    unordered_map<char,char> brackets;
    brackets.insert(make_pair(')','('));
    brackets.insert(make_pair(']','['));
    brackets.insert(make_pair('}','{'));

    stack<char> expression;
    expression.push(s[0]);
    for(int i=1;i<s.length();i++){
        if(brackets.find(s[i]) != brackets.end()){
            if(!expression.empty() && expression.top()==brackets[s[i]]){
                expression.pop();
            }else{
                return false;
            }
        }else{
            expression.push(s[i]);
        }
    }
    return expression.empty();
}
```

Next greater to right, Next greater to left, Next smaller to right, Next smaller to left

This is an important concept that we should know in order to solve subsequent problems related to stacks.

Next Greater To Right:
Consider an array arr=[1, 3, 0, 0, 1, 2, 4]. The output with the next greater element is [3, 4, 1, 1, 2, 4, -1]. The brute force solution would be to find the next greatest for every element using two for loops. Usually in problems where there are two loops required and the inner loop depends on the outer loop, it can be optimized using a stack.

The logic that we will use is, we will traverse the array from the back and also maintain a stack, now what we need to make sure is for every element in the array, the only elements remaining in the stack are the ones greater than that particular number. So whenever we reach a new element, we will pop all those elements from the stack which are smaller than that current element.

```
vector<int> nextGreaterElement(vector<int>& nums) {
    vector<int> NGE(nums.size(),0);
    stack<int> greater;
    for(int i=nums.size()-1; i>=0; i--){
        while(!greater.empty() && greater.top()<=nums[i]){
            greater.pop();
        }
        if(!greater.empty()){
            NGE[i]=greater.top();
        }else{
            NGE[i]=-1;
        }
        greater.push(nums[i]);
    }
    return NGE;
}
```

Next Greater To Left:
This is the exact same approach as the previous one, we just need to change the for loop conditions. Instead of going from right to left, we will go from left to right.

Next Smaller To Left and Next Smaller to Right:
For these two problems, the procedure is exactly the same as the first two, just the way in which we are popping the elements differ. Instead of keeping the larger elements, we will keep the smaller elements.

Online Stock Span

(Link to the problem: https://leetcode.com/problems/online-stock-span/)

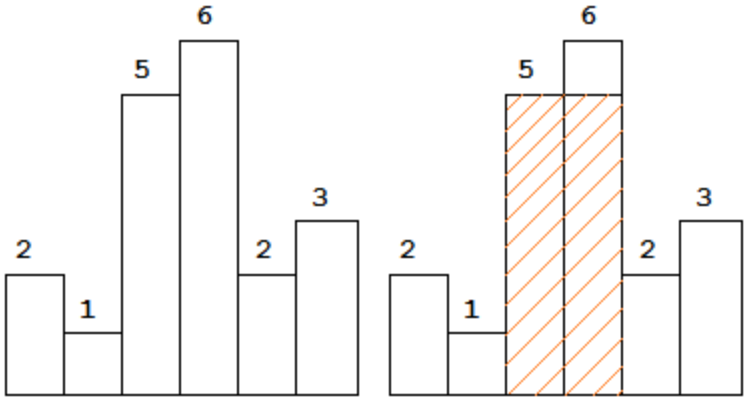
Example:
Input: [100,80,60,70,60,75,85]
Output: [1, 1, 1, 2, 1, 4, 6]

Think, which of the previous four concepts can be used to solve this problem?
We will use **Next Greater To Left**. If we are able to find next greater to the left say at index i, and my current index is j, then the number of consecutive says where the stock price was smaller or equal = ***j-i*.**

Largest Rectangle in Histogram

(Link to the problem: https://leetcode.com/problems/largest-rectangle-in-histogram/)

Example:



The largest rectangle is shown in the shaded area, which has area = 10 units.
Input: [2,1,5,6,2,3]
Output: 10

This is a very interesting problem. So before diving into the explanation, try using the previous approaches to solve this problem.
Alright, so this problem is made up of two sub problems that we have solved before, next smaller left and next smaller right. Now let's say for each bar, we are able to find the largest rectangle it is a part of and then we take the maximum of all them, we will get our answer.

Let's consider two arrays left and right. The left array will comprise the indexes of the next smaller left and right array will comprise the indexes of the next smaller right.

left = [-1, -1, 1, 2, 1, 4]
right = [1, -1, 4, 4, -1, -1]

Now, let's take a look at the bar with height 5. The largest rectangle it can be a part of is between bars of height 1 and height 2, basically the first smaller numbers on either side. If we check out our left and right

arrays, we can find the width of that possible rectangle, width=(right-left-1). And then the area of this rectangle can be calculated by multiplying width with the height of that particular bar, in this case 2(width) x 5(height) = 10 units. We can do this for every bar in the histogram and then take the maximum of all of the rectangles formed.

```
int largestRectangleArea(vector<int>& heights) {
    vector<int> left(heights.size(),0);
    vector<int> right(heights.size(),0);
    stack<pair<int,int>> leftStack;
    stack<pair<int,int>> rightStack;
    int max_area= 0;
    for(int i=0;i<heights.size();i++){
        while(!leftStack.empty() && leftStack.top().first>=heights[i]){
            leftStack.pop();
        }
        if(leftStack.empty()){
            left[i]=-1;
        }else{
            left[i]=leftStack.top().second;
        }
        leftStack.push(make_pair(heights[i],i));
    }
    for(int i=heights.size()-1;i>=0;i--){
        while(!rightStack.empty() && rightStack.top().first>=heights[i]){
            rightStack.pop();
        }
        if(rightStack.empty()){
            right[i]=heights.size();
        }else{
            right[i]=rightStack.top().second;
        }
        rightStack.push(make_pair(heights[i],i));
    }
    for(int i=0;i<heights.size();i++){
        int curr_area = (right[i]-left[i]-1)*heights[i];
        max_area=max(max_area,curr_area);
    }
    return max_area;
}
```

DAY 12 (Stacks and Queues)

Sliding Window Maximum

(Link to the problem: https://leetcode.com/problems/sliding-window-maximum/)

Example:
Input: *nums* = [1,3,-1,-3,5,3,6,7], and *k* = 3
Output: [3,3,5,5,6,7]
Explanation:

```
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7      3
 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      5
 1  3  -1  -3 [5  3  6] 7      6
 1  3  -1  -3  5 [3  6  7]     7
```

For this question we will make use of a deque. A deque is a data structure that can remove and add elements from its front and rear. We will make a deque of size k with its front being the greatest in the window and rear being the smallest. We will be storing the indexes to keep track whether that number lies in the window or not.
For the example *nums* = [1,3,-1,-3,5,3,6,7], and *k* = 3, deque will first look like this, f-> 0 <-r, now we start checking from the rear and removing the numbers smaller than arr[i]. The deque now looks like this, f-> 1 <-r, now when we go to the next number, we see that -1 is smaller than numbers starting from the rear, so we can just push the number to the back, the deque looks like this now, f-> 1 2 <-r. Now when we move to the next window, we encounter -3, and then out deque looks like,

f-> 1 2 3 <-r. Now whenever we are adding new numbers we need to keep checking from the front whether the number lies inside the window or not. Like when we move to the next window, we see that the front of our deque is out of the window, so we remove it, and then the deque looks like f-> 2 3 <-r. Now while adding the new number we keep checking from the rear, then the deque will look like f-> 4 <-r. We can just continue doing the same procedure and get the maximums in every window.
The time complexity is O((n-k).1) , (n-k) cause of the insertions and deletions, and (1) cause of checking the array elements using the indexes.

```
vector<int> maxSlidingWindow(vector<int>& arr, int k) {
    deque<int> Qi(k);
    vector<int> output;

    int i;
    for (i = 0; i < k; ++i) {
        while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();
        Qi.push_back(i);
    }

    for (; i < arr.size(); ++i) {
        output.push_back(arr[Qi.front()]);
        while ((!Qi.empty()) && Qi.front() <= i - k)
            {Qi.pop_front(); }
        while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])
            {Qi.pop_back(); }

    Qi.push_back(i);
}
    output.push_back(arr[Qi.front()]);
    return output;
}
```

Min Stack

(Link to the problem: https://leetcode.com/problems/min-stack/)

Example:
Input: ["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output: [null,null,null,null,-3,null,0,-2]

Explanation: MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top(); // return 0
minStack.getMin(); // return -2

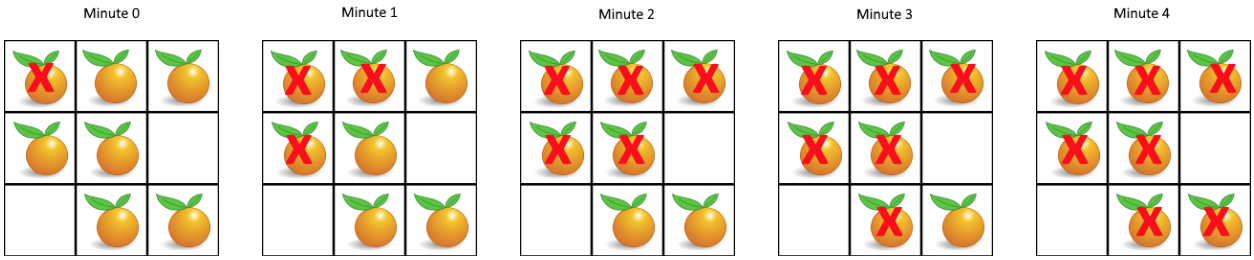
This is actually a very simple problem to implement. We will be making use of two stacks to implement it, one will act as the original stack and the other will keep track of the smallest element present in the stack at that particular moment.
Consider two stacks, original and min. Now we push -2. Original = -2 and min = -2. Now we push 0, original will look like original= -2, 0 but when we are pushing in min, we will check the top of min, if the top is smaller than the new element, we will push the top once again else we will push the new element, so here min = -2, -2. Now we push -3, original = -2, 0, -3 and min = -2, -2, -3. Now when the user asks for the min at any point of time, we can just return the top of stack min.

```
stack<int> main_stack, min_track;
MinStack() {
    main_stack = stack<int>();
    min_track = stack<int>();
}

void push(int x) {
    main_stack.push(x);
    if(min_track.empty() || min_track.top()>x){
        min_track.push(x);
    }else{
        min_track.push(min_track.top());
    }
}

void pop() {
    main_stack.pop();
    min_track.pop();
}

int top() {
    return main_stack.top();
}

int getMin() {
    return min_track.top();
}
```

Rotting Oranges(Using BFS)

(Link to the problem: https://leetcode.com/problems/rotting-oranges/)

Example:



Input: [[2,1,1],[1,1,0],[0,1,1]]
Output: 4
This question is an example of multisource BFS. Check out this video to understand the implementation better.
https://youtu.be/CxrnOTUlNJE

```
bool isValidOrange(vector<vector<int>>& grid, int x, int y){
    return (x>=0 && y>=0 && x<grid.size() && y<grid[0].size());
}
bool anyIsolatedOrange(vector<vector<int>>& grid){
    for(int i=0;i<grid.size();i++){
        for(int j=0;j<grid[0].size();j++){
            if(grid[i][j]==1){
                return true;
            }
        }
    }
    return false;
}
```

```
int orangesRotting(vector<vector<int>>& grid) {
    queue<pair<int,pair<int,int>>> processes;
    for(int i=0;i<grid.size();i++){
        for(int j=0;j<grid[0].size();j++){
            if(grid[i][j]==2){
                processes.push(make_pair(0,make_pair(i,j)));
            }
        }
    }
    int timeConsumed = 0;
    while(!processes.empty()){
        pair<int,pair<int,int>> currentProcess = processes.front();
        timeConsumed = currentProcess.first;
        int x = currentProcess.second.first;
        int y = currentProcess.second.second;
        //top
        if(isValidOrange(grid,x+1,y) && grid[x+1][y]==1){
            grid[x+1][y]=2;
            processes.push(make_pair(timeConsumed+1,make_pair(x+1,y)));
        }
        //bottom
        if(isValidOrange(grid,x-1,y) && grid[x-1][y]==1){
            grid[x-1][y]=2;
            processes.push(make_pair(timeConsumed+1,make_pair(x-1,y)));
        }
        //right
        if(isValidOrange(grid,x,y+1) && grid[x][y+1]==1){
            grid[x][y+1]=2;
            processes.push(make_pair(timeConsumed+1,make_pair(x,y+1)));
        }
        //left
        if(isValidOrange(grid,x,y-1) && grid[x][y-1]==1){
            grid[x][y-1]=2;
            processes.push(make_pair(timeConsumed+1,make_pair(x,y-1)));
        }
        processes.pop();
    }
    if(anyIsolatedOrange(grid)){
        return -1;
    }else{
        return timeConsumed;
    }
}
```

LRU Cache

(Link to the problem: https://leetcode.com/problems/lru-cache/)

This question is a very commonly asked interview question. Check out this video to understand it better. https://youtu.be/Hi5obC_CwIU

```
class LRUCache {
    list<pair<int,int>> cache;
    unordered_map<int,list<pair<int,int>>::iterator> mymap;
    int capacity;
    void refreshLocation(int key,int value){
        cache.erase(mymap[key]);
        cache.push_front(make_pair(key,value));
        mymap[key]=cache.begin();
    }

public:
    LRUCache(int capacity) {
        this->capacity=capacity;
    }

    int get(int key) {
        if(mymap.find(key)!=mymap.end()){
            refreshLocation(key,(*mymap[key]).second);
            return (*mymap[key]).second;
        }
        return -1;
    }

    void put(int key, int value) {
        if(mymap.find(key)!=mymap.end()){
            refreshLocation(key,value);
            return;
        }
        cache.push_front(make_pair(key,value));
        mymap[key]=cache.begin();
        if(mymap.size()>capacity){
            mymap.erase(cache.back().first);
            cache.pop_back();
        }
    }
};
```

DAY 13 (String)

Reverse words in a String

Example:
Input: "the sky is blue"
Output: "blue is sky the"

This is actually a really simple problem, you can approach in two ways:

1. Reverse the entire string, then reverse individual words.
2. Reverse individual words, then reverse the entire string.

```
void reverseWord(string &s, int i, int j){
    while(i<j){
        char temp = s[i];
        s[i++]= s[j];
        s[j--]=temp;
    }
}
string reverseWords(string s) {
    int strlen = s.length();
    int i=0;
    int j=strlen-1;
    //reversing the entire string
    while(i<j){
        char temp = s[i];
        s[i++]= s[j];
        s[j--]=temp;
    }
    i=0,j=0;
    //reversing each word individually
    while(i<strlen){
        while(i<strlen && s[i] == ' ') i++;
        if(i>=strlen) break;
        j=i+1;
        while(j<strlen && s[j] != ' ') j++;
        reverseWord(s,i,j-1);
        i=j;
    }
    return s;
}
```

(pending)Longest Palindromic Substring(DP)

(Link to the problem: https://leetcode.com/problems/longest-palindromic-substring/)

Example:
Input: "babad"
Output: "bab"
Note: "aba" is also a valid answer.

Try this problem after going through the DP questions. After solving the longest common subsequence problems, the first thing that will most probably come to your mind is just reverse the string and apply the longest common substring concept, but that won't work.
Instead what we will do here is create a dp[][] grid, with dimensions length x length, where the rows denote the starting point and columns denote the ending point. Now, what do i mean by that? Let's say you access dp[1][3], this means you're checking whether dp[1][3] or string "aba" is a palindrome or not. We can fill all the diagonal elements with true as every individual character is a palindrome, we can also fill the two character strings like [0,1] or [1,2], by just comparing string[0]==string[1] or string[1]==string[2] and so on. Now lets see if dp[1][3] is a palindrome or "aba" is a palindrome or not, for it to be palindrome string[1] should be equal to string[3] and dp[2][2] should be a palindrome, or for any element dp[start][end] to be a palindrome dp[start+1][end-1] or the inner substring should be a palindrome.
This will result in an $O(n^2)$ solution. But we can do better with something called manacher algorithm.

Implement ATOI function

For a given string convert it into its integer representation.
Example: “324” -> 324
Let’s say the string was ‘3’, to convert this into an integer, we can just subtract ‘0’ from ‘3’ and this will give the difference between the ascii values of the numbers. The difference should lie in the range [0,9], if it doesn’t, then it's not a valid number.
The following code assumes it’s an unsigned integer.

```
int myAtoi(string str) {
    int i=0;
    int result=0;
    while(i<str.length()){
        int difference = str[i++]-'0';
        if(difference>=0 && difference <=9){
            result= (result*10) + difference;
        }else{
            return -1;
        }
    }
    return result;
}
```

Longest Common Prefix

(Link to the problem: https://leetcode.com/problems/longest-common-prefix/)

Example:
Input: ["flower","flow","flight"]
Output: "fl"

There are multiple ways of solving the problem, the way I solved it is by assuming that the first element is the prefix and then iterate through the rest of the array and then comparing the prefix and the current element, if they aren't the same, we reduce the length of the longer string by removing the last character.
In the above example, initially prefix = "flower", when we reach "flow" we notice that they aren't the same and since prefix.length()>element.length(), the new prefix = "flowe", and this goes on till prefix = "flow". If at any point of time prefix = "", we know that there are no common prefixes and we can just return an empty string.

```
string longestCommonPrefix(vector<string>& strs) {
    if(strs.size()==0) return "";
    string prefix = strs[0];
    for(int i=1;i<strs.size();i++){
        while(strs[i].compare(prefix)!=0){
            if(strs[i].length() > prefix.length()){
                strs[i]= strs[i].substr(0,strs[i].length()-1);
            }else if(strs[i].length() < prefix.length()){
                prefix= prefix.substr(0,prefix.length()-1);
            }else{
                strs[i]= strs[i].substr(0,strs[i].length()-1);
                prefix= prefix.substr(0,prefix.length()-1);
            }
        }
        if(prefix.length()==0) break;
    }
    return prefix;
}
```

Rolling Hash | Rabin Karp Algorithm

(Link to the problem: https://leetcode.com/problems/implement-strstr/)

This is a very efficient algorithm for pattern matching. Check out this video to learn it https://youtu.be/qQ8vS2btsxI
//Finish the rolling hash function

DAY 14 (String)

(pending)Prefix Function/Z-function

Example:

(pending)KMP String matching algorithm

Example:
Input: text = "hello", pattern = "ll"
Output: 2

You have to return the starting index of the pattern in the string. You can watch this video to learn the algorithm. https://youtu.be/V5-7GzOfADQ

DAY 15 (Binary Tree)

Inorder Iterative

https://leetcode.com/problems/binary-tree-inorder-traversal/

For writing the iterative inorder traversal, we will make use of a stack.

1. Push current node and go to the left child of root. Do this till you reach a leaf node.
2. If the stack is empty, break.
3. Pop one element from the stack and print it.
4. Go to the right child of the popped node.
5. Repeat till break condition is reached.

```
vector<int> inorderTraversal(TreeNode* root) {
    stack<TreeNode*> nodes;
    vector<int> inorder;
    while(1){
        while(root){
            nodes.push(root);
            root=root->left;
        }
        if(nodes.empty()) break;
        root = nodes.top();
        nodes.pop();
        inorder.push_back(root->val);
        root=root->right;
    }
    return inorder;
}
```

Postorder Iterative

https://leetcode.com/problems/binary-tree-postorder-traversal/

For writing the iterative postorder traversal, we will make use of a stack.

1. Push current node and go to the left child of root. Do this till you reach a leaf node.
2. Now the whole logic behind postorder is that we process the left subtree first, then the right subtree and then finally the root.

Before moving further we check whether the stack is empty or not, if it is then we break from the loop. Now in point 1, we have already processed the left subtree, we now need to check whether we have reached a leaf node or not. We do this by checking if there is a right child of the node at the top of the stack or not. If there is no right child, we just pop the top node and print its value.

3. Now we check whether the popped node is a right child or not, if it's not that means we still have to process the right child. We then push the right child of the element at the top of the stack.
4. If the popped node is a right child, we then just pop the top element from the stack and continue the process.

```
vector<int> postorderTraversal(TreeNode* root) {
    vector<int> postorder;
    stack<TreeNode*> nodes;

    while(1){
        if(root){
            nodes.push(root);
            root=root->left;
        }else{
            if(nodes.empty()) break;
            root = nodes.top()->right;
            if(root == nullptr){
                TreeNode* last = nullptr;
                while(!nodes.empty() && last==nodes.top()->right){
                    last = nodes.top();
                    nodes.pop();
                    postorder.push_back(last->val);
                }
            }
        }
    }

    return postorder;
}
```

Preorder Iterative

https://leetcode.com/problems/binary-tree-postorder-traversal/

For writing the iterative postorder traversal, we will make use of one stack.

1. Initialize a stack and push the root node in it.
2. Run a while loop till the stack is empty. Inside this while loop pop the top element and print it.
3. Push the right child and then the left child of the popped element.

```
vector<int> preorderTraversal(TreeNode* root) {
    vector<int> preorder;
    stack<TreeNode*> nodes;
    if(root == nullptr) return preorder;
    nodes.push(root);
    while(!nodes.empty()){
        TreeNode* topNode = nodes.top();
        nodes.pop();
        preorder.push_back(topNode->val);
        if(topNode->right) nodes.push(topNode->right);
        if(topNode->left) nodes.push(topNode->left);
    }
    return preorder;
}
```

Inorder Recursive

```
void inorderRecursive(TreeNode* root, vector<int> &result){
    if(root==nullptr) return;
    inorderRecursive(root->left, result);
    result.push_back(root->val);
    inorderRecursive(root->right, result);
}

vector<int> inorderTraversal(TreeNode* root) {
    vector<int> inorder;
    inorderRecursive(root, inorder);
    return inorder;
}
```

Postorder Recursive

```
void postorderRecursive(TreeNode* root, vector<int> &result){
    if(root==nullptr) return;
    postorderRecursive(root->left, result);
    postorderRecursive(root->right, result);
    result.push_back(root->val);
}
vector<int> postorderTraversal(TreeNode* root) {
    vector<int> postorder;
    postorderRecursive(root, postorder);
    return postorder;
}
```

Preorder Recursive

```
void preorderRecursive(TreeNode* root, vector<int> &result){
    if(root==nullptr) return;
    result.push_back(root->val);
    preorderRecursive(root->left,result);
    preorderRecursive(root->right,result);
}
vector<int> preorderTraversal(TreeNode* root) {
    vector<int> preorder;
    preorderRecursive(root,preorder);
    return preorder;
}
```

Level Order Traversal

https://leetcode.com/problems/binary-tree-level-order-traversal/

In level order traversal we visit every node from left to right at each level. We will make use of a queue to do this.

```
vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> levelorder;
    if(root==nullptr) return levelorder;
    queue<TreeNode*> qu;
    qu.push(root);
    while(!qu.empty()){
        int len = qu.size();
        vector<int> row;
        while(len--){
            TreeNode* node = qu.front();
            if(node->left) qu.push(node->left);
            if(node->right) qu.push(node->right);
            row.push_back(node->val);
            qu.pop();
        }
        levelorder.push_back(row);
    }
    return levelorder;
}
```

Variable ***len*** is keeping track of the number of nodes at each level.

Spiral Level Order Traversal

https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/

We will make use of a deque to implement this. The idea is simple:
for zig, popback, pushfront, left then right,
for zag, popfront, pushback, right then left.
The same behavior can be implemented using two stacks.

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    vector<vector<int>> zigzag;
    if(root==nullptr) return zigzag;
    deque<TreeNode*> dq;
    dq.push_back(root);
    bool left2right=true;
    while(!dq.empty()){
        int len = dq.size();
        vector<int> row;
        while(len--){
            TreeNode* node = nullptr;
            if(left2right){
                node = dq.front();
                dq.pop_front();
                if(node->left) dq.push_back(node->left);
                if(node->right) dq.push_back(node->right);
            }else{
                node = dq.back();
                dq.pop_back();
                if(node->right) dq.push_front(node->right);
                if(node->left) dq.push_front(node->left);
            }
            row.push_back(node->val);
        }
        zigzag.push_back(row);
        left2right = !left2right;
    }
   return zigzag;
}
```

DAY 16 (Binary Tree)

Height of a Binary Tree

https://leetcode.com/problems/maximum-depth-of-binary-tree/

We will apply something similar to DFS to find the maximum depth of the tree. In the above problem(in the link) the height is Node based height, the below code is for height which is edge based. To change it into node based, just change the base return value to 0 from -1.

```
int maxDepth(TreeNode* root) {
    //Edge based height
    return root?max(maxDepth(root->left),maxDepth(root->right))+1 : -1;
    //Node based height
    //return root?max(maxDepth(root->left),maxDepth(root->right))+1 : 0;
}
```

Diameter of a Binary Tree

https://leetcode.com/problems/diameter-of-binary-tree/

Longest path between any two nodes. The length is based on edges and not nodes.
The maximum diameter can be one of the following:

1. Diameter of left subtree
2. Diameter of right subtree
3. Depth of left subtree + depth of right subtree

```
pair<int,int> dfs(TreeNode* root){
    //return {diameter,depth}
    if(root==nullptr) return {0,0};
    pair<int,int> left= dfs(root->left);
    pair<int,int> right= dfs(root->right);
    int diameter = max({left.first, right.first, left.second + right.second});
    int depth = max(left.second,right.second)+1; //subtree height + 1
    return {diameter,depth};
}

int diameterOfBinaryTree(TreeNode* root) {
    return dfs(root).first;
}
```

Check if two trees are identical or not

https://leetcode.com/problems/same-tree

```
bool isSameTree(TreeNode* p, TreeNode* q) {
    if(p==nullptr and q==nullptr) return true;
    if(p==nullptr or q==nullptr) return false;
    if(p->val != q->val) return false;
    return isSameTree(p->left, q->left) and isSameTree(p->right, q->right);
}
```

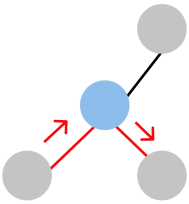
DAY 17 (Binary Tree)

Maximum Path Sum

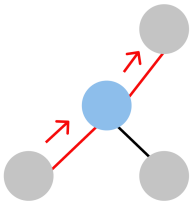
https://leetcode.com/problems/binary-tree-maximum-path-sum/

For every node there are three cases possible:

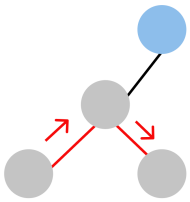
1. The Node is the root of the maximum path.
2. The Node is a part of the straight path of the maximum path.
3. The Node is not a part of the maximum path.



Case 1



Case 2



Case 3

(Red is the maximum sum path and blue is the current node)

For case 1: root_sum = (left + right + root)
For case 2: path_sum = max(max(left, right) + root, root)
For case 3: result = max({result, root_sum, path_sum})
Now let's say we move from that child node to the parent node, we won't return the result because the parent node will lie in the straight path(case 2), and not in the other cases, so we have to return path_sum.

```
int utilityFn(TreeNode* root, int &result){
    if(root==nullptr) return 0;
    int left = utilityFn(root->left, result);
    int right = utilityFn(root->right, result);
    int root_sum = left + right + root->val;
    int path_sum = max(max(left, right) + root->val, root->val);
    result = max({result, root_sum, path_sum});
    return path_sum;
}
int maxPathSum(TreeNode* root) {
    int result = INT_MIN;
    utilityFn(root, result);
    return result;
}
```

Lowest Common Ancestor of two nodes

https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree

The logic is pretty easy to understand, we first search for both the nodes, as soon as we find the required node we return to its parent. Now, if a node receives both the required nodes from its left and right side, it is the Lowest Common Ancestor. Check out this video to understand it better: https://youtu.be/F-_1sbnPbWQ

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if(root==nullptr) return nullptr;
    if(root==p or root==q) return root; //since a node can be a descendant of itself

    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);

    if(left && right){
        return root;
    }else{
        return left?left:right;
    }
}
```

Symmetric Tree

https://leetcode.com/problems/symmetric-tree/

For a tree to be symmetric, it should be a mirror image of itself.

```
bool isMirror(TreeNode* t1, TreeNode* t2){
    if(t1==nullptr and t2==nullptr) return true;
    if(t1==nullptr or t2==nullptr) return false;
    return (t1->val==t2->val)
        && isMirror(t1->left,t2->right)
        && isMirror(t1->right, t2->left);

}

bool isSymmetric(TreeNode* root) {
    return isMirror(root, root);
}
```

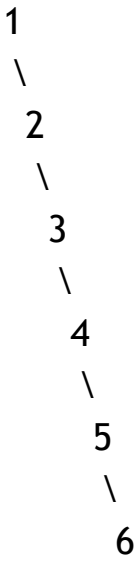
Flatten Binary Tree to a Linked List

https://leetcode.com/problems/flatten-binary-tree-to-linked-list/

For example, given the following tree:

```
  1
  / \
 2   5
/ \   \
3   4   6
```

The flattened tree should look like:



On Day 6 under linked list we had solved a similar problem in which we had flattened a doubly linked list into a single linked list. We had made use of a stack to store the pending processes. To solve this problem, we will use the exact same approach. We will consider the left node as the child processes and the right node as the next process.

```
void flatten(TreeNode* root) {
    stack<TreeNode*> pending;
    while(root){
        if(root->left){
            if(root->right){
                pending.push(root->right);
            }
            root->right=root->left;
            root->left=nullptr;
        }
        if(!(root->right) and !pending.empty()){
            root->right=pending.top();
            pending.pop();
        }
        root=root->right;
    }
}
```

DAY 18 (Binary Search Tree)

Populating Next Right Pointers in Each Node

https://leetcode.com/problems/populating-next-right-pointers-in-each-node/

For this problem we will use the concept of level order traversal to find the solution.

```
Node* connect(Node* root) {
    if(root==nullptr) return root;
    queue<Node*> qu;
    qu.push(root);
    while(!qu.empty()){
        int len = qu.size();
        Node* cur = nullptr;
        while(len--){
            if(cur!=nullptr) cur->next=qu.front();
            cur = qu.front();
            qu.pop();
            if(cur->left) qu.push(cur->left);
            if(cur->right) qu.push(cur->right);
        }
    }
    return root;
}
```

Search in Binary Search Tree

https://leetcode.com/problems/search-in-a-binary-search-tree/

This is a very basic question based on the fundamental concept of BST.

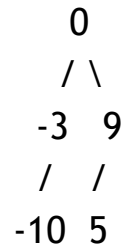
```
TreeNode* searchBST(TreeNode* root, int val) {
    if(root==nullptr) return root;
    if(root->val==val) return root;
    return (root->val)<val?searchBST(root->right,val):searchBST(root->left,val);
}
```

Convert Sorted Array to Binary Search Tree

https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/

We will apply a similar approach as binary search here, but recursively. The key thing to remember is that the BST should be height balanced.

Example: Given the sorted array: [-10,-3,0,5,9], one possible answer is: [0,-3,9,-10,null,5], which represents the following height balanced BST:




```
TreeNode* constructBST(vector<int>& nums, int start, int end){
    if(start>end) return nullptr;
    int mid = start + (end - start)/2;
    TreeNode* node = new TreeNode(nums[mid]);
    node->left = constructBST(nums, start, mid-1);
    node->right= constructBST(nums, mid+1, end);
    return node;
}
TreeNode* sortedArrayToBST(vector<int>& nums) {
    if(nums.empty()) return nullptr;
    return constructBST(nums, 0, nums.size()-1);
}
```

Lowest Common Ancestor in Binary Search Tree

https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/

The way we will be solving this problem is by searching for the point where the two nodes diverge. This is easy because we have a BST. It'll become more clear once you see the code.

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    int val = root->val;
    if(val > p->val and val > q->val){
        return lowestCommonAncestor(root->left, p, q);
    }
    if(val < p->val and val < q->val){
        return lowestCommonAncestor(root->right, p, q);
    }
    //Now since the p and q and not part of the same subtree,
    //we know this is the point where they diverge
    return root;
}
```

Try dry running the code with a BST, you'll understand how it's working.

DAY 19 (Binary Search Tree)

Find K-th smallest element in a BST

https://leetcode.com/problems/kth-smallest-element-in-a-bst/

We know that inorder traversal on a BST gives us values which are monotonically increasing, hence we just need to find the first k elements.

```
int kthSmallest(TreeNode* root, int k) {
    stack<TreeNode*> nodes;
    while(1){
        while(root){
            nodes.push(root);
            root=root->left;
        }
        root = nodes.top();
        nodes.pop();
        if(--k == 0) return root->val;
        root = root->right;
    }
}
```

Morris Traversal Solution with O(1) space

//pending

Find a pair with a given sum in BST

https://leetcode.com/problems/two-sum-iv-input-is-a-bst/

We have solved a similar two sum problem before with a hashmap.

```
bool findTarget(TreeNode* root, int k) {
    unordered_set<int> visitedNodes;
    stack<TreeNode*> nodes;
    while(1){
        while(root){
            nodes.push(root);
            root=root->left;
        }
        if(nodes.empty()) break;
        root = nodes.top();
        nodes.pop();
        if(visitedNodes.find(k-(root->val)) != visitedNodes.end()){
            return true;
        }
        visitedNodes.insert(root->val);
        root=root->right;
    }

    return false;
}
```

DAY 20 (Heaps)

Find K-th largest element in an array

https://leetcode.com/problems/kth-largest-element-in-an-array/

We will consider a minHeap of size k, by the time we are done traversing the array the element at the top of the minHeap will be the K-th largest element.

```
int findKthLargest(vector<int>& nums, int k) {
    priority_queue<int, vector<int>, greater<int>> minHeap;
    for(int i=0;i<nums.size();i++){
        minHeap.push(nums[i]);
        if(minHeap.size()>k) minHeap.pop();
    }
    return minHeap.top();
}
```

K Closest numbers

Say we need to find the K closest numbers to a number N. So to solve this problem we will make use of a maxHeap, and enter the elements with respect to the difference between every element of the array and N.

Example:
Input: [5,6,7,8,9] k=3 N=7
Output: [7,6,8]

```
vector<int> findClosestElements(vector<int>& arr, int k, int x) {
    priority_queue<pair<int,int>> pq; // <difference, number>
    for(int i=0;i<arr.size();i++){
        pq.push(make_pair(abs(arr[i]-x),arr[i]));
        if(pq.size()>k){
            pq.pop();
        }
    }
    vector<int> result;

    while(!pq.empty()){
        result.push_back(pq.top().second);
        pq.pop();
    }
    return result;
}
```

Top K Frequent Numbers

https://leetcode.com/problems/top-k-frequent-elements/

We will consider a minHeap of size k.

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    unordered_map<int,int> mp;
    priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> minHeap;
    for(int i=0;i<nums.size();i++){
        mp[nums[i]]++;
    }
    for(auto i=mp.begin();i!=mp.end();i++){
        minHeap.push({i->second,i->first});
        if(minHeap.size()>k){
            minHeap.pop();
        }
    }
    vector<int> result;
    while(!minHeap.empty()){
        result.push_back(minHeap.top().second);
        minHeap.pop();
    }
    return result;

}
```

K Closest Points to Origin

https://leetcode.com/problems/k-closest-points-to-origin/

We will consider a maxHeap of size k and insert with respect to the distance between the point and the origin.

```
vector<vector<int>> kClosest(vector<vector<int>>& points, int K) {
    priority_queue<pair<int,int>> maxHeap;
    for(int i=0;i<points.size();i++){
        int x=points[i][0];
        int y=points[i][1];
        int distance = pow(x,2) + pow(y,2);
        maxHeap.push(make_pair(distance,i));
        if(maxHeap.size()>K) maxHeap.pop();
    }
    vector<vector<int>> result;
    while(!maxHeap.empty()){
        result.push_back(points[maxHeap.top().second]);
        maxHeap.pop();
    }
    return result;
}
```

Minimum Cost of joining the ropes

https://leetcode.com/discuss/interview-question/344677/Amazon-or-Online-Assessment-2019-or-Min-Cost-to-Connect-Ropes

We will build a minHeap and enter all the elements, till the time we dont have only one rope remaining in the heap we will keep selecting the top two smallest and merging them and add the new rope to the heap.

```
int mergeRopes(vector<int>& ropes) {
    priority_queue<int, vector<int>, greater<int>> minHeap;
    int res = 0;
    for(int i:ropes){
        minHeap.push(i);
    }
    while(minHeap.size()>1){
        int first = minHeap.top();
        minHeap.pop();
        int second = minHeap.top();
        minHeap.pop();
        res+= first+second;
        minHeap.push(first+second);
    }
    return res;
}
```

DAY 21 (Graphs)

BFS

First move horizontally and visit all the nodes in the current layer, and then move to the next layer. For any graph traversal we will keep track of the visited nodes.

Pseudo Code:

```
BFS(G,s) //s is the starting node
        Queue.enqueue(s)
        Mark s as visited
        while(Queue not empty)
                V = Queue.dequeue()
                For all neighbours W of V in graph G
                        If W is not visited
                                Queue.enqueue(W)
                                Mark W visited
```

```
void BFS(int s){
    queue<int> pending;
    vector<bool> visited(m_v,false);
    pending.push(s);
    visited[s]= true;
    while(!pending.empty()){
        int v = pending.front();
        pending.pop();
        cout<< v << endl;
        for(int u:m_adj[v]){
            if(!visited[u]){
                pending.push(u);
                visited[u]=true;
            }
        }
    }
};
```

DFS

We start at one node and go as deep as possible. Explore as far as possible in a branch and then backtrack.

Pseudo Code:

```
DFS_Iterative(G,s) //s is the starting node
        St <- stack
        St.push(s)
        Mark s as visited
        While St is not empty
                U <- St.pop()
                For all adjacent V of U
                        If V is not visited
                                St.push(V)
                                Mark V as visited
```

```cpp
class Graph{
    int m_v;
    vector<vector<int>> m_adj;
    void DFS_recursive(int s, vector<bool>& visited){
        visited[s]=true;
        cout<< s <<endl;
        for(int u:m_adj[s]){
            if(!visited[u]){
                DFS_recursive(u, visited);
            }
        }
    };

    public:
        Graph(int v):m_v(v), m_adj(v){};

        void addEdge(int u, int v){
            m_adj[u].push_back(v);
            //If it's undirected
            //m_adj[v].push_back(u);
        };

        void DFS(){
            vector<bool> visited(m_v,false);
            for(int i=0;i<visited.size();i++){
                if(!visited[i]){
                    DFS_recursive(i,visited);
                }
            }
        };
};
```

Applications of BFS and DFS

Here we will see various applications/questions where we can use the above traversals.

DFS:
Find Minimum Spanning Tree
Cycle Detection
Path between 2 Nodes
Topological Sorting
Bipartite Check
SCC
Maze Problems

BFS:
Shortest Path in Undirected Graph
Cycle Detection
Bipartite Check
Path between 2 Nodes
Minimum Spanning Tree in Undirected Graph
Social Networking Websites
GPS Navigation
Network Broadcasting
Connected Components

Clone a Graph

(Link to the problem: https://leetcode.com/problems/clone-graph/)

Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]
Explanation: There are 4 nodes in the graph.
1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val=4).
2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val=3).
3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val=4).
4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val=3).

We need to create a deep copy of a graph. To solve this problem we will use DFS and I'll explain how.
First we just create a copy of the node(new_node) that is passed to us(passed_node) by creating a new node. We create a visited map and mark new_node as visited. Then we run a DFS on passed_node, if the neighbour node is visited, then we insert it into the node's neighbour list.

```
Node* cloneGraph(Node* node) {
    if(!node) return nullptr;
    unordered_map<int,Node*> visited;
    return cloneNode(node, visited);
}

Node* cloneNode(Node* node, unordered_map<int,Node*>& visited){
    Node* new_node = new Node(node->val);
    visited.insert({new_node->val,new_node});

    for(Node* n:node->neighbors){
        //To check if we've already cloned this node or not
        auto it = visited.find(n->val);

        if(it==visited.end()){//Not yet cloned
            //Trigger clone of its neighbor
            Node* cloned_node = cloneNode(n, visited);
            //Once cloned, add the neighbor to the neighbors list
            new_node->neighbors.push_back(cloned_node);
        }else{
            //Neighbor already cloned, add to the neighbors list
            new_node->neighbors.push_back(it->second);
        }
    }
    return new_node;
}
```

Cycle Detection(Directed)

We will do a DFS here. Like how we had maintained a visited array which stored boolean values, instead what we will do is store states of the nodes in it. There are three possible states: UNVISITED, INSTACK, VISITED. Let's say you have applied a DFS on a node, now that node is still in stack, so it's state is Instack, now while the DFS is happening, suppose you encounter a node which has an edge connected to the original node whose state is INSTACK, that means there is a cycle formed. Reason being from the original node we have come to this node and we are not yet done visiting it and hence it forms a cycle.

```
enum node_states_en{
    UNVISITED,
    INSTACK,
    VISITED
};

class Graph{
    int m_v;
    vector<vector<int>> m_adj;
    bool DFS_recursive_MOD(int s, vector<node_states_en>& visited){
        visited[s]=INSTACK;
        cout<< s <<endl;
        for(int u:m_adj[s]){
            if(visited[u]==INSTACK) return true; //Has a cycle
            if(visited[u]==UNVISITED and DFS_recursive_MOD(u, visited)) return true;
        }
        visited[s]=VISITED;
        return false;
    };

    public:
        Graph(int v):m_v(v), m_adj(v){};

        void addEdge(int u, int v){
            m_adj[u].push_back(v);
            //If it's undirected
            //m_adj[v].push_back(u);
        };

        bool hasCycle(int s){
            vector<node_states_en> visited(m_v, UNVISITED);
            for(int i=0;i<visited.size();i++){
                if(visited[i]==UNVISITED and DFS_recursive_MOD(i,visited)) return true;
            }
            return false;
        };
```

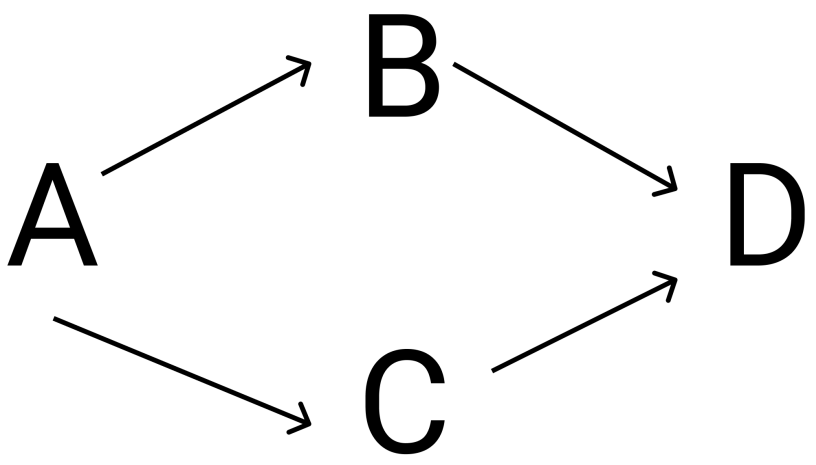
Cycle Detection(Undirected)

This solution for this will be similar to Directed Graph Cycle detection, we just need to make sure that the node that is already visited and we are trying to visit it again is not the parent node of the current node. We can just do this by passing the parent node along with the neighbor node.

```
class Graph{
    //Vertices
    int m_v;
    //Adj List
    vector<vector<int>> m_adj;

    //For cycle detection
    bool DFS_recursive_MOD2(int s, vector<bool>& visited, int parent){
        visited[s]=true;
        for(int u:m_adj[s]){
            if(visited[u] and u != parent) return true;
            if(!visited[u] and DFS_recursive_MOD2(u, visited, s)) return true;
        }
        return false;
    };

    public:
        Graph(int v):m_v(v), m_adj(v){};

        void addEdge(int u, int v){
            m_adj[u].push_back(v);
            //If it's undirected
            //m_adj[v].push_back(u);
        };

        bool hasCycleUndirected(){
            vector<bool> visited(m_v, false);
            for(int i=0;i<visited.size();i++){
                if(!visited[i] and DFS_recursive_MOD2(i,visited,-1)) return true;
            }
            return false;
        };
};
```

Topological Sort

For a Directed Acyclic Graph, topological sort is linear ordering of vertices such that for every edge (u,v) vertex u comes before vertex v. Topological Sort is not unique. Let's checkout an example.



The topological sort for this DAG can be [A,B,C,D] or [A,C,B,D]. If you notice, A comes before B and C, and B and C come before D.
Topological Sort is useful for the following:

1. Build Systems
2. Task Scheduling
3. Course Scheduling
4. Package Manager

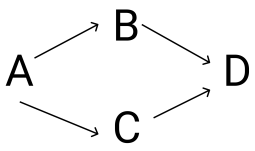
The way we will solve this is by applying DFS on the DAG, and when we reach a node which has no more unvisited neighbors, we push that node in a stack. For the above example, one possible status of the stack by the end of DFS can be: bottom->[D,B,C,A]<-top. Now we can just start popping them and printing them and we will get [A,C,B,D] and this satisfies the topological sort conditions.

```
class Graph{
    //Vertices
    int m_v;
    //Adj List
    vector<vector<int>> m_adj;

    //For Topological Sort DFS recursion
    void TS_recursive(int s, vector<bool>& visited, stack<int>& S){
        visited[s]=true;
        for(int u:m_adj[s]){
            if(!visited[u]){
                TS_recursive(u, visited, S);
            }
        }
        S.push(s);
    };

    public:
        Graph(int v):m_v(v), m_adj(v){};

        void addEdge(int u, int v){
            m_adj[u].push_back(v);
            //If it's undirected
            //m_adj[v].push_back(u);
        };

        void Topo_Sort(){
            vector<bool> visited(m_v,false);
            stack<int> S;
            for(int i=0;i<visited.size();i++){
                if(!visited[i]){
                    TS_recursive(i,visited,S);
                }
            }
            while(!S.empty()){
                cout<<S.top()<<" ";
                S.pop();
            }
        };
};
```

Course Scheduling

(Link to the problem: https://leetcode.com/problems/course-schedule-ii/)

Example:
Input: 4, [[1,0],[2,0],[3,1],[3,2]]
Output: [0,1,2,3] or [0,2,1,3]



This is what the graph of the above courses(A:0,B:1,C:2,D:3) look like.

This is a mixture of two problems; topological sort and cycle detection in a directed graph.

```
class Solution {
    bool dfs(int u, vector<vector<int>>& adj, vector<int>& order, vector<int>& visited){
        visited[u]=1; //instack

        for(int v:adj[u]){
            if(visited[v]==1) return true;//Cycle Found
            if(visited[v]==0 and dfs(v,adj,order,visited)) return true;//Cycle Found
        }

        visited[u]=2; //visited
        order.push_back(u);
        return false;//No cycle found
    }
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> adjList(numCourses);
        //Building Adjacency List
        for(int i=0;i<prerequisites.size();i++){
            adjList[prerequisites[i][1]].push_back(prerequisites[i][0]);
        }

        // You can also use a stack, but since we have to return a vector
        // i have used vector here. We will just reverse it in the end and return.
        vector<int> order;

        //0:unvisited, 1:instack, 2:visited
        vector<int> visited(numCourses, 0);

        for(int i=0;i<visited.size();i++){
            if(visited[i]==0 && dfs(i, adjList, order, visited)) return {};
        }

        reverse(order.begin(),order.end());
        return order;
    }
};
```

Number of Islands

(Link to the problem: https://leetcode.com/problems/number-of-islands/)

Example:

Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]

Output: 3

We will employ a simple dfs and wherever we get a '1' we make it 0.

```
void dfs(int x, int y, vector<vector<char>>& grid){
    grid[x][y]=0;
    //Bottom
    if(x+1<grid.size() and grid[x+1][y]=='1') dfs(x+1,y,grid);
    //Top
    if(x-1>=0 and grid[x-1][y]=='1') dfs(x-1,y,grid);
    //Right
    if(y+1<grid[0].size() and grid[x][y+1]=='1') dfs(x,y+1,grid);
    //Left
    if(y-1>=0 and grid[x][y-1]=='1') dfs(x,y-1,grid);
}
int numIslands(vector<vector<char>>& grid) {
    int islands=0;

    for(int i=0;i<grid.size();i++){
        for(int j=0;j<grid[0].size();j++){
            if(grid[i][j]=='1'){
                dfs(i,j,grid);
                islands++;//We have encountered one island
            }
        }
    }

    return islands;
}
```

Surrounded Regions

(Link to the problem: https://leetcode.com/problems/surrounded-regions/)

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

All we have to do here is find a region that lies on the boundary of the board because those regions can't be surrounded by water, and once we find a region like that we find all the regions adjacent to it by applying DFS and change it to some character say 'N'. Once we are done with this, we loop through the entire board again and change all the 'O' to 'X', and all the 'N' to 'O'.

```
void dfs(int x, int y, vector<vector<char>>& board){
    board[x][y]='N';
    //Bottom
    if(x+1<board.size() and board[x+1][y]=='O') dfs(x+1,y,board);
    //Top
    if(x-1>=0 and board[x-1][y]=='O') dfs(x-1,y,board);
    //Right
    if(y+1<board[0].size() and board[x][y+1]=='O') dfs(x,y+1,board);
    //Left
    if(y-1>=0 and board[x][y-1]=='O') dfs(x,y-1,board);
}
void solve(vector<vector<char>>& board) {
    for(int i=0;i<board.size();i++){
        for(int j=0;j<board[0].size();j++){
            if(i==0||j==0||i==board.size()-1||j==board[0].size()-1){
                //Region is at the boundary hence can't be surrounded by water
                if(board[i][j]=='O'){
                    //Run DFS to find all the adjacent regions
                    dfs(i,j,board);
                }
            }
        }
    }
    //Convert 'N' back to 'O' and Remove all surrounded regions
    for(int i=0;i<board.size();i++){
        for(int j=0;j<board[0].size();j++){
            if(board[i][j]=='O'){
                board[i][j]='X';
            }else if(board[i][j]=='N') board[i][j]='O';
        }
    }

}
```

DAY 22 (Graphs)

Find all paths from source to target

https://leetcode.com/problems/all-paths-from-source-to-target/

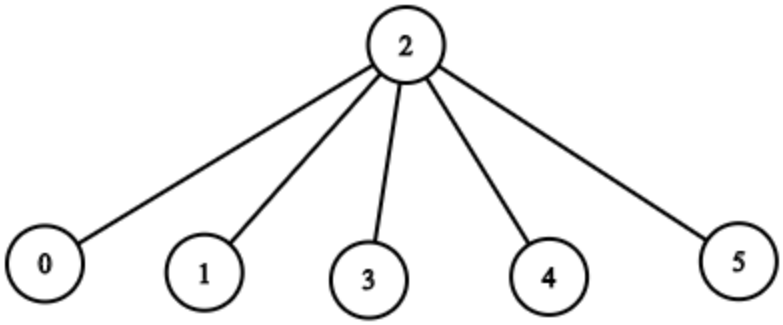
This problem is actually a simple DFS, here we don’t need to maintain a visited array and also since it's a DAG we don’t have to worry about any cycles.

```
void dfs(vector<vector<int>>& graph, vector<vector<int>>& result, vector<int>& path, int u){
    path.push_back(u);
    if(u==graph.size()-1){
        //We have reached the destination
        result.push_back(path);
    }else{
        for(int v:graph[u]){
            dfs(graph, result, path, v);
        }
    }
    path.pop_back(); //Go one step back in the path
}

vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
    vector<vector<int>> result;
    vector<int> path;
    dfs(graph, result, path, 0); //Because all the paths start at 0
    return result;
}
```

Time Needed to Inform All Employees

(Link to the problem: https://leetcode.com/problems/time-needed-to-inform-all-employees/)

Example:
Input: n = 6, headID = 2, manager = [2,2,-1,2,2,2], informTime = [0,0,1,0,0,0]
Output: 1
Explanation: The head of the company with id = 2 is the direct manager of all the employees in the company and needs 1 minute to inform them all.
The tree structure of the employees in the company is shown.



First we need to make an adjacency list to represent the above tree. Note that it's a tree and hence no cycles. This is a simple dfs application, dry run the code and it'll make good sense.

```
class Solution {
    int dfs(vector<vector<int>>& adjL, vector<int>& informTime, int headID){
        int maxTime=0;
        //Visit all nodes of the current head
        for(auto i:adjL[headID]){
            maxTime = max(maxTime,dfs(adjL,informTime,i));
        }
        return maxTime + informTime[headID];
    }
public:
    int numOfMinutes(int n, int headID, vector<int>& manager, vector<int>& informTime) {
        vector<vector<int>> adjL(n);
        for(int i=0;i<manager.size();i++){
            if(manager[i]!=-1){
                adjL[manager[i]].push_back(i);
            }
        }
        return dfs(adjL,informTime,headID);
    }
};
```

Keys and Rooms

(Link to the problem: https://leetcode.com/problems/keys-and-rooms/)

Example:
Input: [[1],[2],[3],[]]
Output: true
Explanation:
We start in room 0, and pick up key 1.
We then go to room 1, and pick up key 2.
We then go to room 2, and pick up key 3.
We then go to room 3. Since we were able to go to every room, we return true.

This is yet another simple dfs question, we just need to check the visited array in the end for any unvisited nodes.

```
class Solution {
    void dfs(int s, vector<bool>& visited, vector<vector<int>>& rooms){
        visited[s]=true;
        for(int i:rooms[s]){
            if(!visited[i]) dfs(i,visited,rooms);
        }
    }
public:
    bool canVisitAllRooms(vector<vector<int>>& rooms) {
        vector<bool>visited(rooms.size(),false);
        visited[0]=true;
        for(int i:rooms[0]){
            if(!visited[i]) dfs(i,visited,rooms);
        }
        for(bool i:visited){
            if(!i) return false;
        }
        return true;
    }
};
```

Union Find Algorithm

Refer to this video to understand how union and find operations work on DSUs https://youtu.be/eTaWFhPXPz4
Identify if problems talk about finding groups or components. You can check out this link to understand the operations better
http://www.goodtecher.com/union-find/

Friend Circles

(Link to the problem: https://leetcode.com/problems/friend-circles/)

Example:
Input:
[[1,1,0],
 [1,1,0],
 [0,0,1]]
Output: 2
Explanation: The 0_{th} and 1_{st} students are direct friends, so they are in a friend circle.
The 2_{nd} student himself is in a friend circle. So return 2.

Note that this question can be solved using DFS, but I wanted to show you how we can use union find for grouping type questions.

In the above example, we basically get two sets {0,1} and {2}. So basically when we are looping through the matrix if(M[i][j]==1 and i!=j) then make a group. If you've gone through the video we will use the union find data structure where find will tell us whether two elements are part of the same group or not, and the union will merge two groups into one group.

```cpp
class Solution {
    vector<int>parent;
    int circleCount;

    int find(int x) {
        //If the parent is not the root, then set it's value to its absolute parent
        return parent[x] == x ? x : find(parent[x]);
    }

    void uni(int a, int b){
        int groupA = find(a);
        int groupB = find(b);
        if(groupA != groupB){
            parent[groupA]=groupB; //Merging the two groups
            circleCount--; //Reduce the # of friend circles cause of merging
        }
    }

public:
    int findCircleNum(vector<vector<int>>& M) {
        int n = M.size(); //Nodes 0,1,2
        parent.resize(n, 0);
        for (int i = 0; i < n; i++) parent[i] = i;

        //Initial the number of circles = individual people
        circleCount = n;

        for(int i=0;i<M.size();i++){
            for(int j=0;j<M[0].size();j++){
                if(M[i][j]==1 and i!=j){
                    uni(i,j);
                }
            }
        }

        return circleCount;
    }
};
```

Redundant Connection

(Link to the problem: https://leetcode.com/problems/redundant-connection/)

Example:
Input: [[1,2], [2,3], [3,4], [1,4], [1,5]]
Output: [1,4]
Explanation: The given undirected graph will be like this:
5 - 1 - 2
| |
4 - 3

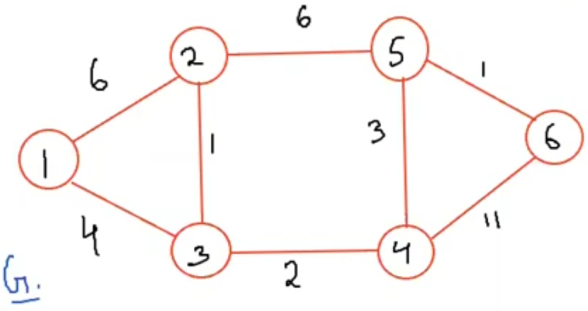
The important thing to note is that the graph is originally a tree and they've added an extra edge here by mistake that needs to be removed. Whenever two nodes share an edge or share an **absolute** parent that is just a redundant edge. In the above example when the first 3 pairs were processed and then [1,4] was added, [1,2,3,4] already shared the same absolute parent, so there was no use of adding [1,4] because there is already a path from 1 to 4.

```
class Solution {
    vector<int> parent;
    int find(int x){
        return parent[x] == x ? x : find(parent[x]);
    }
public:
    vector<int> findRedundantConnection(vector<vector<int>>& edges) {
        int n = edges.size();
        parent.resize(n+1,0);

        for(int i=0;i<parent.size();i++){
            parent[i]=i;
        }
        vector<int> res(2,0);
        for(int i=0;i<edges.size();i++){
            int x = find(edges[i][0]);
            int y = find(edges[i][1]);
            if(x!=y){
                //Dont share the same absolute parent
                parent[y]=x;
            }else{
                res[0] = edges[i][0];
                res[1] = edges[i][1];
            }
        }
        return res;
    }
};
```

DAY 23 (Graphs)

Dijkstra Algorithm

The Dijkstra algorithm is used for finding the shortest path from a source node to every other node.



What is the shortest path from node 1 to node 2? It's 1->3->2. Dijkstra makes use of the greedy algorithm to solve it. It initially takes an array and marks all the distances to infinity and mark the source as 0. So for the above graph the distance matrix will initially look like [x,0,inf,inf,inf,inf,inf]. Now we take a set data structure and insert a pair of <distance,node> into it. Initially we insert <0,1> since 1 is the source node. Now, while the set is not empty we will take the pair that has the smallest distance and store it in some temp_pair and then remove that temp_pair from the set. Now, iterate through all the neighbours of the node stored in temp_pair. In the above example say the first node that we encounter is 2, now i will take the distance stored in temp_pair, because it is the distance I've travelled to reach that node in temp_pair and add it to the distance between the node in temp_pair and the neighbour, in this case 0+6=6. Now i check if this distance is smaller than the distance stored in the distances array at index 2. If it is smaller I will erase any previous occurrences of the pair which included node 2 and insert the pair <6,2> into the set.

I will do the same for all the remaining nodes. Finally we will be left with a distances array containing all the shortest distances from the source node.

```
int main() {
    //{node,distance}
    vector<vector<pair<int,int>>> adjL = {
        {},
        {{2,6},{3,4}},
        {{1,6},{3,1},{5,6}},
        {{1,4},{2,1},{4,2}},
        {{3,2},{5,3},{6,11}},
        {{2,6},{4,3},{6,1}},
        {{5,1},{4,11}}
        };
    vector<int> distances(adjL.size(),INT_MAX);
    int source = 1;
    distances[source]=0;
    //{distance,node}
    set<pair<int,int>> s;
    s.insert({0,source});
    while(!s.empty()){
        pair<int,int> temp = *s.begin();
        s.erase(s.begin());
        int u = temp.second;
        for(auto i:adjL[u]){
            int v = i.first;
            int distance = i.second;
            if(distances[v]>distances[u]+distance){
                if(distances[v]!=INT_MAX) s.erase(s.find(make_pair(distances[v],v)));
                distances[v] = distances[u]+distance;
                s.insert({distances[v],v});
            }
        }
    }
}
```

(Pending) Minimum Spanning Tree

DAY 24 (Dynamic Programming)

Max Product Subarray

https://leetcode.com/problems/maximum-product-subarray/

Example:
Input: [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.

This is a slight modification to Kadane's Algorithm. Here we have to keep track of the prevMax and prevMin. Let's say the array was [-2,3,-2,4]. Till index 1, the maximum is 3 and the minimum is -2. Now if we didn't keep track of the previous minimum, we wouldn't be able to calculate the maximum value till index 2, which is -2x3-2=12. So basically our maximum is the maximum of the three: prevMax*a[i], prevMin*a[i], or a[i] itself. And to calculate the minimum till now, just take the minimum of the three. Try dry running the code on [-1, 6, 2, 0, 7, 9].

```
int maxProduct(vector<int>& nums) {
    if(nums.empty()) return -1;
    int curMax=nums[0], curMin=nums[0], prevMax=nums[0], prevMin=nums[0], ans=nums[0];

    for(int i=1;i<nums.size();i++){
        curMax= max({prevMax*nums[i],prevMin*nums[i],nums[i]});
        curMin= min({prevMax*nums[i],prevMin*nums[i],nums[i]});
        ans = max(ans,curMax);
        prevMax=curMax;
        prevMin=curMin;
    }

    return ans;
}
```

Longest Increasing Subsequence

https://leetcode.com/problems/longest-increasing-subsequence/

Example:
Input: [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.
A subsequence has the same relative order as the original array.

$O(n^2)$ Approach:
In this approach we will have an array L which will define the longest subsequence possible if the longest subsequence ends with A[i] where A is the original array.
For example:
L[0] = 1 ([10])
L[1] = 1 ([9])
L[2] = 1 ([2])
L[3] = 2 ([2,5]) or L[2]+1
L[4] = 2 ([2,3]) or L[2]+1
L[5] = 3 (max([2,3,7],[5,7])) or max(L[4]+1, L[3]+1)
L[6] = 4 (max([10,101],[9,101],[5,7,101],[2,3,7,101]))
L[7] = 4 (max([10,18],[9,18],[5,7,18],[2,3,7,18]))
So the longest increasing subsequence possible is max(elements in L[]).

```
int lengthOfLIS(vector<int>& nums) {
    if(nums.empty()) return 0;
    vector<int> L(nums.size(),1);
    for(int i=1;i<nums.size();i++){
        for(int j=0;j<i;j++){
            if(nums[j]<nums[i] and L[i]<L[j]+1){
                L[i]=L[j]+1;
            }
        }
    }
    return *max_element(L.begin(),L.end());
}
```

O(nlogn) Approach:

This is a really good video to understand the logic behind this O(nlogn) approach. https://youtu.be/22s1xxRvy28

```
int lengthOfLIS(vector<int>& nums) {
    vector<int> dp(nums.size());
    int len=0;
    for(int num:nums){
        int i= lower_bound(dp.begin(),dp.begin()+len,num)-dp.begin();
        dp[i]=num;
        if(i==len) len++;
    }
    return len;
}
```

In C++, lower_bound returns the index at which the number is >= the number that you are required to find.

Types Knapsack Problem

1. Fractional Knapsack (Greedy Problem)
   https://www.geeksforgeeks.org/fractional-knapsack-problem/
2. 0/1 Knapsack
   https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
3. Unbounded Knapsack
   https://www.geeksforgeeks.org/unbounded-knapsack-repetition-items-allowed/

Go through all the above links to understand the concept behind Knapsack problems. After that we will see how to implement it in different ways.

0/1 Knapsack(Recursive):
I/P: wt=[1,3,4,5], val=[1,4,5,7], W=7
O/P: Maximum profit

Now,there are two situations available for an item, either the weight of the item is less than or equal to the maximum weight, or the weight of the item is greater than the maximum weight. For the first case we have two options, either we include it in the bag, or we don't include it in the bag, for the second case we only have one option of not including in the bag.
Time Complexity: $O(2^n)$

```
int knapsack(vector<int> wt, vector<int> val, int W){
    //Base Case: Think of the smallest valid input
    if(wt.empty() or W==0) return 0;

    //To decide whether to add a particular item or not
    //In recursion we have to reduce the input size
    if(wt[wt.size()-1]<=W){
        int weight = wt[wt.size()-1];
        int value = val[val.size()-1];
        wt.pop_back();
        val.pop_back();
        return max(value+knapsack(wt,val,W-weight),knapsack(wt,val,W));
    }else{
        wt.pop_back();
        val.pop_back();
        return knapsack(wt,val,W);
    }
}
```

0/1 Knapsack(Memoization):

Once you have the recursive code ready, memoization becomes really simple. Our first step would be to define a grid/table of n*m and decide what the n and m parameters are going to be. To do this we need to identify the parameters that are changing, in this case the size of the weight array and the remaining weight W is changing.

Now since we need to store the values till N and W, let's make a grid of dp[N+1][W+1]. After this we will just initialize the grid with -1 in each cell. Now every time before I call the recursive function, I will check whether that value exists in the grid dp or not.

```cpp
int knapsack(vector<int> wt, vector<int> val, int W, vector<vector<int>>& dp,int n){
    //Base Case: Think of the smallest valid input
    if(n==0 or W==0) return 0;
    //If already calculated for that particular n and W
    if(dp[n][W]!=-1) return dp[n][W];


    //To decide whether to add a particular item or not
    //In recursion we have to reduce the input size
    if(wt[n-1]<=W){
        int weight = wt[n-1];
        int value = val[n-1];
        dp[n][W]=max(value+knapsack(wt,val,W-weight,dp,n-1),knapsack(wt,val,W,dp,n-1));
        return dp[n][W];
    }else{
        dp[n][W]=knapsack(wt,val,W,dp,n-1);
        return dp[n][W];
    }
    return 0;
}
```

Problems based on 0/1 Knapsack:

1. Subset sum problem
2. Equal sum partition
3. Count of subset sum
4. Minimum subset sum difference
5. Target sum
6. Number of subsets with the given difference

The trick to identify knapsack based problems is to identify if there is a pattern where you have to decide whether to choose an item or not, along with some capacity present.

Subset sum problem

Example:
Input: [2,3,7,8,10] sum=11
Output: true
We have to find whether there exists a subset with sum 11. Do you notice the similarity with the knapsack problem? Imagine the given array is the weight array.
Here we will make a grid dp[5+1][11+1] for the above example.
What is our smallest sub problem? If we have an array of size 0 and the required sum is also 0, we have to either return true or false, in this (0,0) case we will return true. Similarly, for any size of the array if the required sum is 0, it will always return true, just make an empty subset. Now, we have an array of size 0 and the required sum is more than 0, then we can never have a valid subset.

T	F	F	F	F	F	F	F	F	F	F	F
T	.	.	.	.	.	.	.	.	.	.	.
T	.	.	.	.	.	.	.	.	.	.	.
T	.	.	.	.	.	.	.	.	.	.	.
T	.	.	.	.	.	.	.	.	.	.	.
T	.	.	.	.	.	.	.	.	.	.	.

Our required answer will lie at dp[n][sum]

```cpp
int main() {
    vector<int> arr = {2,3,7,8,10};
    int sum = 11;
    vector<vector<bool>> dp(arr.size()+1,vector<bool>(sum+1));
    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            if(i==0) dp[i][j]=false;
            if(j==0) dp[i][j]=true;
        }
    }
    for(int i=1;i<dp.size();i++){
        for(int j=1;j<dp[0].size();j++){
            if(arr[i-1]>j){
                dp[i][j]= dp[i-1][j];
            }
            if(arr[i-1]<=j){
                dp[i][j]= dp[i-1][j] || dp[i-1][j-arr[i-1]];
            }
        }
    }

     for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            cout<<dp[i][j]<<"\t";
        }
        cout<<endl;
    }
    cout<<boolalpha<<dp[arr.size()][sum];
}
```

Equal Sum Partition

https://leetcode.com/problems/partition-equal-subset-sum/

Example:
Input: [1, 5, 11, 5]
Output: true
Explanation: The array can be partitioned as [1, 5, 5] and [11].

Let's see how we can reduce this problem to the regular subset sum problem. If we add all the elements and calculate the total sum, and then if the sum is odd, we can say that it is not possible to partition it into two **equal** parts, whereas if the sum is even, there **might** be a subset with the required sum. For the above example, sum=22. Since this sum is even, there is a chance that we might have a subset of 11. So our goal is to find whether there exists a subset of sum = sum/2, if we have found a subset that satisfies that, we can say that we can partition this array into equal subset.

```cpp
bool canPartition(vector<int>& nums) {
    //Check if it's possible to partition equally
    int sum = 0;
    for(int num:nums) sum+=num;
    if(sum%2!=0) return false;

    sum/=2;

    //Apply subset sum problem for the above sum
    vector<vector<bool>> dp(nums.size()+1,vector<bool>(sum+1));
    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            if(i==0) dp[i][j]=false;
            if(j==0) dp[i][j]=true;
        }
    }

    for(int i=1;i<dp.size();i++){
        for(int j=1;j<dp[0].size();j++){
            if(nums[i-1]<=j){
                dp[i][j]= dp[i-1][j] or dp[i-1][j-nums[i-1]];
            }else{
                dp[i][j]= dp[i-1][j];
            }
        }
    }

    return dp[nums.size()][sum];
}
```

DAY 25 (Dynamic Programming)

Count subsets with a given sum

Example:
Input: [2,3,5,6,8,10] sum=10
Output: 3 - {2,8}, {2,3,5}, {10}

This question is very similar to the regular subset sum problem, instead of returning true or false we just have to return the count. The main difference in the code arises in the initialization of the dp[][] grid, and instead of doing an 'or' operation when we have a choice of choosing a number or not, we do an 'addition' operation inorder to calculate all the possible values.

```
//Now count the number of subsets with the above sum
vector<vector<int>> dp(nums.size()+1,vector<int>(sum+1));
for(int i=0;i<dp.size();i++){
    for(int j=0;j<dp[0].size();j++){
        if(i==0) dp[i][j]=0;
        if(j==0) dp[i][j]=1;
    }
}

for(int i=1;i<dp.size();i++){
    for(int j=1;j<dp[0].size();j++){
        if(nums[i-1]<=j){
            dp[i][j]= dp[i-1][j] + dp[i-1][j-nums[i-1]];
        }else{
            dp[i][j]= dp[i-1][j];
        }
    }
}
cout<<dp[nums.size()][sum];
```

Minimum Subset Sum Difference

Example:
Input: [1,5,6,11]
Output: 1

We need to partition the array into two subsets such that the absolute difference between the individual sums of the subset is minimum. Here, {5,6} and {11,1}, and the minimum difference is 12-11=1. Let's take a smaller array [1,2,7]. The minimum difference possible is {7}-{1,3}=4.

Now, consider we partition this array into two sums s1 and s2 and assume that **s1** is **smaller** than s2. What do you think is the possible range where s1 and s2 might lie? Let's look at the extreme subsets, either the subset can be a null subset {}, or a subset can contain all the numbers {1,2,7}. So s1 and s2 has to lie somewhere between 0--------(1+2+7), or in the range [0,10]. Now, how can we simplify this further? Do we really need to calculate both s1 and s2? We know that **s2=range-s1**, so instead of minimizing **(s2-s1)** we just need to minimize **(range-2s1)**. So we have removed one extra variable. Since we have assumed that s1 is smaller than s2, we can restrict s1 to be strictly <=range/2 (10/2=5). So, s1 can either be {0,1,2,3,4,5}.
Now, think how we can involve the regular subset sum problem here? If you remember, each cell of the dp[][] array tells us whether it's possible to have a subset of some sum x for the given array of size n. Here since we have to include all the elements, we essentially just have to look at the last row of the dp[][] array.
Checkout the code to understand it better.

```cpp
int main() {
    vector<int> arr = {1,5,6,11};
    int sum = 0;
    for(int num:arr) sum+=num;
    vector<vector<bool>> dp(arr.size()+1,vector<bool>(sum+1));
    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            if(i==0) dp[i][j]=false;
            if(j==0) dp[i][j]=true;
        }
    }
    for(int i=1;i<dp.size();i++){
        for(int j=1;j<dp[0].size();j++){
            if(arr[i-1]>j){
                dp[i][j]= dp[i-1][j];
            }
            if(arr[i-1]<=j){
                dp[i][j]= dp[i-1][j] || dp[i-1][j-arr[i-1]];
            }
        }
    }
    //Storing Valid values of S1
    vector<int> valid;
    for(int j=0;j<=(sum/2);j++){
        if(dp[dp.size()-1][j]) valid.push_back(j);
    }
    int minDiff = INT_MAX;
    for(int i=0;i<valid.size();i++){
        minDiff = min(minDiff, sum-(2*valid[i])); //Minimizing (range - 2*s1)
    }
    cout<<endl;
    cout<<minDiff;
    return 0;
}
```

Count the number of subset partitions with the given difference

Example:
Input: [1,1,2,3] diff=1
Output: 3

This is actually a really simple question if you've understood the previous problems. Let's first look at what we have. We have to divide the array into two subsets s1 and s2 such that s1-s2=diff.

s1 - s2 = diff ------ (1)
s1 + s2 = total_sum ------ (2)

If we do (1) + (2), we get

2*s1 = diff + total_sum
s1 = (diff + total_sum)/2

So basically, we need to count the number of subsets with the sum 's1'.

```cpp
int main() {
    vector<int> arr = {1,1,2,3};
    int diff=1;
    int total_sum = 0;
    for(int num:arr) total_sum+=num;
    int s1 = (diff+total_sum)/2;
    vector<vector<int>> dp(arr.size()+1,vector<int>(s1+1));
    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            if(i==0) dp[i][j]=0;
            if(j==0) dp[i][j]=1;
        }
    }
    for(int i=1;i<dp.size();i++){
        for(int j=1;j<dp[0].size();j++){
            if(arr[i-1]>j){
                dp[i][j]= dp[i-1][j];
            }
            if(arr[i-1]<=j){
                dp[i][j]= dp[i-1][j] + dp[i-1][j-arr[i-1]];
            }
        }
    }
    cout<<dp[arr.size()][s1];
    return 0;
}
```

Target Sum

https://leetcode.com/problems/target-sum/

Example:
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.
This problem is almost the same as the previous problem, we are just partitioning the array into two subsets and subtracting them or taking all the numbers with the same sign in the same subset and taking a difference of them.
We just need to make two adjustments here:

1. Take care of 0's. To consider the extra subsets of 0', if the number of 0's is n, then the total number of subsets is increased by 2^n number of times.
2. Check if (total_sum+diff) is even or not, if it's not even, then it's not possible to partition the array.

```
int findTargetSumWays(vector<int>& nums, int S) {
    int total_sum = 0;
    int zeros=0;
    for(int num:nums) {
        if(num==0) zeros++;
        total_sum+=num;
    };
    if(S>total_sum) return 0;
    if((S+total_sum)%2!=0) return 0;
    int s1 = (S+total_sum)/2;
    vector<vector<int>> dp(nums.size()+1,vector<int>(s1+1));
    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            if(i==0) dp[i][j]=0;
            if(j==0) dp[i][j]=1;
        }
    }
    for(int i=1;i<dp.size();i++){
        for(int j=1;j<dp[0].size();j++){
            if(nums[i-1]==0){
                //We will take care of the subsets of 0 in the end
                dp[i][j]=dp[i-1][j];
            }else if(nums[i-1]>j){
                dp[i][j]= dp[i-1][j];
            }else{
                dp[i][j]= dp[i-1][j] + dp[i-1][j-nums[i-1]];
            }
        }
    }
    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            cout<<dp[i][j]<<"\t";
        }
        cout<<endl;
    }
    return pow(2,zeros)*dp[nums.size()][s1];
}
```

Unbounded Knapsack

The only difference between 0/1 Knapsack and Unbounded Knapsack is that we have multiple occurrences of the items. And the only part where we will change the code is when we have to make the decision to take the item or exclude it, if we have excluded it, it's processed and don't have to look at that again, but if we have to include it, we have to keep using the item. So instead of:

dp[i][j]= max(val[i-1] + dp[i-1][j-wt[i-1]], dp[i-1][j])

we have

dp[i][j]= max(val[i-1] + dp[i][j-wt[i-1]], dp[i-1][j])

Problems based on Unbounded Knapsack:

1. Rod Cutting
2. Coin Change-| (Maximum number of ways)
3. Coin Change-|| (Minimum number of coins)
4. Maximum Ribbon Cut

Rod Cutting

Example:

length	1	2	3	4	5	6	7	8
price	3	5	8	9	10	17	17	20

We are given three variables as inputs: price[], length[], size_of_rod(N). Let's see how this is analogous to our regular knapsack problem.

N->W(the max capacity), price[]->val[] and length[]->wt[]

Now that we know it's a type of knapsack problem, how do we know whether it's unbounded or 0/1. If you look at the problem, you'll notice that we can **repeat** the number of times a rod of the same length can be cut, for example if N=8, we can cut it as 3-3-2.

```
int main() {
    vector<int> profit = {1,5,8,9,10};
    int N = 4;
    vector<int> length;
    for(int i=1;i<=profit.size();i++){
        length.push_back(i);
    }
    vector<vector<int>> dp(profit.size()+1,vector<int>(N+1));
    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            if(i==0 || j==0) dp[i][j]=0;
        }
    }
    for(int i=1;i<dp.size();i++){
        for(int j=1;j<dp[0].size();j++){
            if(length[i-1]>j){
                dp[i][j]= dp[i-1][j];
            }
            if(length[i-1]<=j){
                dp[i][j]= max(dp[i-1][j],   profit[i-1]+ dp[i][j-length[i-1]]);
            }
        }
    }
```

```
    //Backtracking to find the cuts
    int x = profit.size(), y = N;
    vector<int> rods;
    while (x > 0 && y > 0) {
        if (dp[x - 1][y] == dp[x][y])
            x--;
        else if (dp[x - 1][y] >= profit[x - 1] + dp[x][y - length[x - 1]])
            x--;
        else {
            cout<<"Including item "<<x<<" with value = "<<profit[x - 1]<<" and length = "<<length[x - 1]<<endl;
            rods.push_back(length[x - 1]);
            y -= length[x - 1];
        }
    }
    cout<<endl;
    // Marking points at which cut has to be made.
    int cut = 0;
    for (int a : rods) {
        cut += a;
        cout<<"Cut the rod at x = "<< cut<<endl;
    }
    cout<<endl;
    cout<<"Max Profit = "<<dp[profit.size()][N];
    return 0;
}
```

DAY 26 (Dynamic Programming)

Coin Change Problem:Maximum Number of ways

https://www.hackerrank.com/challenges/coin-change/problem

What previously solved question does this resemble? We have to find all the subsets or coin combinations such that their sum equals to a given sum. Here since we can use one coin more than once, hence it is an unbounded knapsack problem.

```
long getWays(int n, vector<long> c) {
    vector<vector<long>>dp(c.size()+1,vector<long>(n+1));
    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            if(i==0) dp[i][j]=0;
            if(j==0) dp[i][j]=1;
        }
    }
    for(int i=1;i<dp.size();i++){
        for(int j=1;j<dp[0].size();j++){
            if(j<c[i-1]){
                dp[i][j]=dp[i-1][j];
            }else{
                dp[i][j]=dp[i-1][j]+dp[i][j-c[i-1]];
            }
        }
    }
    return dp[c.size()][n];
}
```

Minimum Number of coins

https://leetcode.com/problems/coin-change/

Example:
Input: [1,2,3] sum=5
Output: 2 (2+3)

Here again we will initialize the dp with dimensions [n+1] x [sum+1] where n is the size of the array.
Now, imagine we have 0 coins and we need some sum ‘x’. To get the sum x from 0 coins, theoretically we’ll need an infinite number of coins, hence we will initialize it with INT_MAX-1. ‘-1’ because whenever change is possible we will add ‘+1’ so to make sure we don't get integer overflow, or basically for safety purposes we have to subtract ‘-1’.
And if we have 1 or more than 1 coin, we will need 0 coins to get a sum 0.
Now, consider you have 1 coin and let’s say that coin is 3, and the sum required is 5, and we know that this isn’t possible, so we fill INT_MAX-1 here, or we just need to check if the sum is divisible by the first coin, if it is then we will fill sum/(first_coin_value), else if it’s not we will fill INT_MAX-1.

```
int coinChange(vector<int>& coins, int amount) {
   vector<vector<int>> dp(coins.size()+1,vector<int>(amount+1));
    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            if(j==0) dp[i][j]=0;
            if(i==0) dp[i][j]=INT_MAX-1;
            if(i==1){
                if(j%coins[0]==0){
                    dp[i][j]=j/coins[0];
                }else{
                    dp[i][j]=INT_MAX-1;
                }
            }
        }
    }

    //Code for unbounded knapsack
    for(int i=1;i<dp.size();i++){
        for(int j=1;j<dp[0].size();j++){
            if(coins[i-1]<=j){
                dp[i][j]= min(dp[i-1][j], 1 + dp[i][j-coins[i-1]]);
            }else{
                dp[i][j]= dp[i-1][j];
            }
        }
    }
    if(dp[coins.size()][amount]==INT_MAX-1){
        return -1;
    }else{
        return dp[coins.size()][amount];
    }

}
```

Longest Common Subsequence

https://leetcode.com/problems/longest-common-subsequence/

Example:
Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.

Checkout this detailed video to understand the approach
https://youtu.be/43P0xZp3FU4

```cpp
int longestCommonSubsequence(string text1, string text2) {
    vector<vector<int>> dp(text1.size()+1,vector<int>(text2.size()+1));
    for(int i=0;i<=text1.size();i++){
        dp[i][0]=0;
    }
    for(int i=0;i<=text2.size();i++){
        dp[0][i]=0;
    }
    for(int i=1;i<=text1.size();i++){
        for(int j=1;j<=text2.size();j++){
            if(text1[i-1]==text2[j-1]) dp[i][j]=dp[i-1][j-1]+1;
            else dp[i][j]= max(dp[i-1][j],dp[i][j-1]);
        }
    }
    string lcs="";
    int i=text1.size();
    int j=text2.size();
    //To print the LCS by backtracking
    while(i>0 && j>0){
        if(text1[i-1]==text2[j-1]){
            lcs=text1[i-1]+lcs;
            i--;
            j--;
        }else{
            if(dp[i-1][j]>=dp[i][j-1]){
                i--;
            }else{
                j--;
            }
        }
    }
    cout<<lcs;
    return dp[text1.size()][text2.size()];
}
```

Longest Common Substring

Example:
Input: text1 = "abcde", text2 = "abfce"
Output: 2 (“ab”)

This problem is just a variation of Longest common subsequence.

```
int main() {
    string text1 = "abcde", text2 = "abcfce";
    vector<vector<int>> dp(text1.size()+1,vector<int>(text2.size()+1));

    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            if(i==0 || j==0) dp[i][j]=0;
        }
    }

    for(int i=1;i<dp.size();i++){
        for(int j=1;j<dp[0].size();j++){
            if(text1[i-1]==text2[j-1]){
                dp[i][j]=dp[i-1][j-1]+1;
            }else{
                //We have hit a discontinuity
                //Start a new substring
                dp[i][j]=0;
            }
        }
    }
    int lcs=INT_MIN;
    for(int i=0;i<dp.size();i++){
        for(int j=0;j<dp[0].size();j++){
            lcs=max(dp[i][j],lcs);
        }
    }
    cout<<lcs;
    return 0;
}
```

Shortest Common Supersequence

https://leetcode.com/problems/shortest-common-supersequence/

Example:
Input: str1 = "abac", str2 = "cab"
Output: "cabac"
Explanation:
str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".
str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".
The answer provided is the shortest such string that satisfies these properties.

This problem is just another variation of Longest Common Subsequence. I have found the common subsequence between the two strings, and then using 3 pointers, I have merged them.

```
string shortestCommonSupersequence(string text1, string text2) {
    vector<vector<int>> dp(text1.size()+1,vector<int>(text2.size()+1));

    for(int i=0;i<=text1.size();i++){
        for(int j=0;j<=text2.size();j++){
             if(i==0 || j==0) dp[i][j]=0;
        }
    }

    for(int i=1;i<=text1.size();i++){
        for(int j=1;j<=text2.size();j++){
            if(text1[i-1]==text2[j-1]) {
                dp[i][j]=dp[i-1][j-1]+1;
            }
            else dp[i][j]= max(dp[i-1][j],dp[i][j-1]);
        }
    }
    string lcs="";
    int i=text1.size();
    int j=text2.size();
    //To get the LCS by backtracking
    while(i>0 && j>0){
        if(text1[i-1]==text2[j-1]){
            lcs=text1[i-1]+lcs;
            i--;
            j--;
        }else{
            if(dp[i-1][j]>=dp[i][j-1]){
                i--;
            }else{
                j--;
            }
        }
    }

    i=0;//for text1
    j=0;//for text2
    int k=0;//for lcs
```

```
    string super="";

    while(i<text1.size() and j<text2.size() and k<lcs.size()){
        if(text1[i]==lcs[k] and text2[j]==lcs[k]){
            //Overlapping character
            super+=lcs[k];
            i++;
            j++;
            k++;
        }else if(text1[i]==lcs[k]){
            //Common character in text1
            super+=text2[j];
            j++;
        }else if(text2[j]==lcs[k]){
            //Common character in text2
            super+=text1[i];
            i++;
        }else{
            super+= text1[i];
            super+= text2[j];
            i++;
            j++;
        }
    }
    while(i<text1.size()){
        super+=text1[i];
        i++;
    }
    while(j<text2.size()){
        super+=text2[j];
        j++;
    }

    return super;
}
```

Minimum number of Insertions or Deletions to change string a to b

Example:
Input: text1 = "heap", text2 = "pea"
Output: Deletions: 2 Insertions: 1

So if you notice, to convert “heap” to “pea” we can first delete ‘h’ and ‘p’ in “heap” and then insert ‘p’ in the first string. Do you notice which string remains unchanged? Yes, it’s the longest common subsequence that remains unchanged. Now think how will you find the number of deletions and insertions if you have the lcs?

Deletions = text1.size() - lcs.size();
Insertions = text2.size() - lcs.size();

```
int main() {
    string text1 = "a", text2 = "abcd";
    vector<vector<int>> dp(text1.size()+1,vector<int>(text2.size()+1));

    for(int i=0;i<=text1.size();i++){
        for(int j=0;j<=text2.size();j++){
            if(i==0 || j==0) dp[i][j]=0;
        }
    }

    for(int i=1;i<=text1.size();i++){
        for(int j=1;j<=text2.size();j++){
            if(text1[i-1]==text2[j-1]) {
                dp[i][j]=dp[i-1][j-1]+1;
            }
            else dp[i][j]= max(dp[i-1][j],dp[i][j-1]);
        }
    }
    int lcs_length = dp[text1.size()][text2.size()];
    int deletions = text1.size()-lcs_length;
    int insertions = text2.size()-lcs_length;
    cout<<"Deletions: "<<deletions<<"\tInsertions: "<<insertions;
    return 0;
}
```

DAY 27 (Dynamic Programming)

Longest Palindromic Subsequence

https://www.hackerrank.com/challenges/coin-change/problem

Example:
Input: "bbbab"
Output: 4

Can you try and figure out how we can use the concept of LCS here? We have only one string, how will we get the second string? What if the second string is a function of the first string?

For this problem, we just need to find the longest common subsequence between the first string and the reverse of the first string, and we will get the longest palindromic subsequence!

```
int longestPalindromeSubseq(string s) {
    string text1=s;
    reverse(s.begin(),s.end()); //Reversing the string
    string text2=s;
    vector<vector<int>> dp(text1.size()+1,vector<int>(text2.size()+1));

    for(int i=0;i<=text1.size();i++){
        for(int j=0;j<=text2.size();j++){
             if(i==0 || j==0) dp[i][j]=0;
        }
    }

    for(int i=1;i<=text1.size();i++){
        for(int j=1;j<=text2.size();j++){
            if(text1[i-1]==text2[j-1]) {
                dp[i][j]=dp[i-1][j-1]+1;
            }
            else dp[i][j]= max(dp[i-1][j],dp[i][j-1]);
        }
    }

    return dp[text1.size()][text2.size()];
}
```

Minimum number of deletions to make a string palindrome

Example:
Input: “agbcba”
Output: 1

If you’ve understood the problem based on conversion of one string to another with minimum insertions and deletions, this should be easy for you to solve. We just need to find the longest palindromic subsequence and subtract its length from the original string.

```
int main() {
    string text1 = "agbcba";
    string text2 = text1;
    reverse(text2.begin(),text2.end());
    vector<vector<int>> dp(text1.size()+1,vector<int>(text2.size()+1));

    for(int i=0;i<=text1.size();i++){
        for(int j=0;j<=text2.size();j++){
            if(i==0 || j==0) dp[i][j]=0;
        }
    }

    for(int i=1;i<=text1.size();i++){
        for(int j=1;j<=text2.size();j++){
            if(text1[i-1]==text2[j-1]) {
                dp[i][j]=dp[i-1][j-1]+1;
            }
            else dp[i][j]= max(dp[i-1][j],dp[i][j-1]);
        }
    }
    int lcs_length = dp[text1.size()][text2.size()];
    int deletions = text1.size()-lcs_length;
    cout<<"Deletions: "<<deletions;
    return 0;
}
```

Minimum Insertion Steps to Make a String Palindrome

https://leetcode.com/problems/minimum-insertion-steps-to-make-a-string-palindrome/

Example:
Input: s = "zzazz"
Output: 0
Explanation: The string "zzazz" is already palindrome we don't need any insertions

Can you somehow relate this to the minimum number of deletions required to make a string palindrome? Let's say our string was "aebcbda", to convert it into a palindrome the minimum number of deletions required is 2(remove 'e' and 'd'). What does this mean? There are two characters which don't have a pair because of which the string isn't a palindrome. So we just need to adjust these two characters to get a palindrome. So basically minimum number of insertions = minimum number of deletions.

```
int minInsertions(string s) {
    string text1 = s;
    string text2 = text1;
    reverse(text2.begin(),text2.end());
    vector<vector<int>> dp(text1.size()+1,vector<int>(text2.size()+1));

    for(int i=0;i<=text1.size();i++){
        for(int j=0;j<=text2.size();j++){
                if(i==0 || j==0) dp[i][j]=0;
        }
    }

    for(int i=1;i<=text1.size();i++){
        for(int j=1;j<=text2.size();j++){
            if(text1[i-1]==text2[j-1]) {
                dp[i][j]=dp[i-1][j-1]+1;
            }
            else dp[i][j]= max(dp[i-1][j],dp[i][j-1]);
        }
    }
    int lcs_length = dp[text1.size()][text2.size()];
    int insertions = text1.size()-lcs_length;

    return insertions;
}
```

Matrix Chain Multiplication(MCM)

This is another format of DP problems. The types of problem that come under this category are:

1. MCM
2. Printing MCM
3. Evaluate expression to true/boolean parenthesization
4. min/max value of an expression
5. Palindrome partitioning
6. Scramble String
7. Egg Dropping problem

How to identify MCM based questions?

You are usually given a string or an array. Say you have an array [.....], now we consider two pointers i and j like this [i…….j] and then another pointer k that breaks [i,j] like this [i…….k….j] now we have two separate problems [i,k] and [k+1,j] which are called temporary answers, using these two temp answers we solve the main problem and get the answer. We basically have to apply some function on the temp answers to get the original answer. Once we start solving the problems, you'll understand what I mean by breaking the problem at k.

Pseudocode for general format of MCM questions:

```
int solve(int arr[], int i, int j){
        //Base Condition
        if(i>j){ //Depends on the question
                return 0;
        }
        for(int k=i; k<j; k++){
                //Calculate temp ans
                tempAns = solve(arr,i,k) + solve(arr,k+1,j);//Depends on the question
                ans = some_function(tempAns);
        }
}
```

Matrix Chain Multiplication Implementation

Example:
Input: arr = [40,20,30,10,30]
Output: 0

Let's first see what we mean by MCM, suppose you have three matrices A,B, and C. You can either multiply as (AB)C or A(BC), now our goal is to minimize the cost, or the # of multiplications. So for a matrix A1:10x30 and a matrix A2:30x40 the total cost on A1A2 = 10x30x40=12000. Hence for ABC we have to minimize the cost.
Let's see how this translates to our question. We are given an array of size 5, so there will be n-1 or 4 matrices formed. How?

A_1:40x20 **A_2:20x30** **A_3:30x10** **A_4:10x30**

Or simple A_i=arr[i-1]xarr[i]. Now we need to break it at k to figure out where we should be putting brackets for example, $(A1A2A3)_k$(A4) or $(A1A2)_k$(A3A4). Let's take the case $(A1A2)_k$(A3A4):

40x20 20x30 30x10 10x30

cost1=40x20x30 cost2=30x10x30

40x30 30x30

cost3=40x30x30

temp_ans = cost1 + cost2 + cost3

We will get cost1 and cost2 by recursively calling the solve fn, but we need to manually calculate cost3. If you backtrack the values in cost3 you will notice cost3 = arr[i-1] x arr[k] x arr[j]. And finally we just have to take the minimum of all the temp_ans.

Recursive Solution:

```cpp
int solve(vector<int>& arr, int i, int j){
    if(i>=j){// '=' because array of 1 element is also invalid
        return 0;
    }
    int minimum = INT_MAX;
    for(int k=i; k<=j-1; k++){//j-1 because if k=j, we will get an empty array from k+1 or right partition
        //Calculate temp ans
        int cost1 = solve(arr,i,k);
        int cost2 = solve(arr,k+1,j);
        int cost3 = arr[i-1]*arr[k]*arr[j];
        int tempAns = cost1+cost2+cost3;
        //ans = some_function(tempAns);
        minimum = min(minimum,tempAns);
    }
    return minimum;
}

int main() {
    vector<int> arr = {10, 20, 30, 40, 30};
    cout<< solve(arr, 1, arr.size()-1); //1 because we know Ai= arr[i-1]xarr[i]
}
```

Now we will memoize this solution with a simple few steps.

Memoized Solution:

```cpp
int static dp[1001][1001];
int solve(vector<int>& arr, int i, int j){
    if(i>=j){// '=' because array of 1 element is also invalid
        return 0;
    }
    if(dp[i][j]!=-1) return dp[i][j];
    int minimum = INT_MAX;
    for(int k=i; k<=j-1; k++){//j-1 because if k=j, we will get an empty array from k+1 or right partition
        //Calculate temp ans
        int cost1 = solve(arr,i,k);
        int cost2 = solve(arr,k+1,j);
        int cost3 = arr[i-1]*arr[k]*arr[j];
        int tempAns = cost1+cost2+cost3;
        //ans = some_function(tempAns);
        minimum = min(minimum,tempAns);
    }
    return dp[i][j]=minimum;
}

int main() {
    vector<int> arr = {10, 20, 30, 40, 30};
    memset(dp,-1,sizeof(dp));
    cout<< solve(arr, 1, arr.size()-1); //1 because we know Ai= arr[i-1]xarr[i]
}
```

Burst Balloons

https://leetcode.com/problems/burst-balloons/

Example:
Input: [3,1,5,8]
Output: 167
Explanation: 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

This question is an exact implementation of MCM, the only difference is the function that we apply on the temp_ans. Instead of minimizing the cost, we will maximize it, and insert 1 at the beginning and at the end because it states in the question that if we are bursting the left or right most balloons, the cost = 1 x burst x right or left x burst x 1.

```
int dp[1001][1001];

int solve(vector<int>& arr, int i, int j){
if(i>=j){// '=' because array of 1 element is also invalid
    return 0;
}
if(dp[i][j]!=-1) return dp[i][j];
int maximum = INT_MIN;
for(int k=i; k<=j-1; k++){//j-1 because if k=j, we will get an empty array from k+1 or right partition
    //Calculate temp ans
    int cost1 = solve(arr,i,k);
    int cost2 = solve(arr,k+1,j);
    int cost3 = arr[i-1]*arr[k]*arr[j];
    int tempAns = cost1+cost2+cost3;
    //ans = some_function(tempAns);
    maximum = max(maximum,tempAns);
}
return dp[i][j]=maximum;
}

int maxCoins(vector<int>& nums) {
    nums.insert(nums.begin(),1);
    nums.push_back(1);
    memset(dp,-1,sizeof(dp));
    return solve(nums, 1, nums.size()-1);
}
```

DAY 28 (Dynamic Programming)

(Pending)Egg Drop

https://leetcode.com/problems/super-egg-drop/

Example:
Input: K = 1, N = 2
Output: 2
Explanation:
Drop the egg from floor 1. If it breaks, we know with certainty that F = 0.
Otherwise, drop the egg from floor 2. If it breaks, we know with certainty that F = 1.
If it didn't break, then we know with certainty F = 2.
Hence, we needed 2 moves in the worst case to know what F is with certainty

We will apply MCM here where k will denote the floor where we drop the egg from. What do you think the base condition will be for this?
Let's say you have e eggs and f floors, if e =1, then in the worst case we have to check all the floors from the bottom one-by-one, so we can just return f. Now if f=1, then we just need to check once, so we can again return f.

Now, let's try and figure out how we will break it into sub problems. Suppose we drop at k. Now there are two cases: 1) It breaks, 2)It doesn't break. So if the egg breaks at k, we are left with (e-1) eggs, and if it breaks at k, we know that it will break at any floor greater than k, this means that we just have to check for (k-1) floors with (e-1) eggs. But, if the egg doesn't break, then the number of eggs remains the same but now we know that we don't have to check any floor less than k, so we

only have to check (f-k) floors. Let's first see the recursive code and then the memoized code.

```
int solve(int e, int f){
    if(e==1 or f==1 or f==0) return f;
    int minimum = INT_MAX;
    for(int k=1; k<=f;k++){
        //Max because we need to find the worst case
        int temp_ans = 1 + max(solve(e-1, k-1), solve(e, f-k));
        int minimum = min(minimum, temp_ans);
    }
    return minimum;
}
int superEggDrop(int K, int N) {
    return solve(K,N);
}
```

(Pending) Palindrome Partitioning

https://leetcode.com/problems/palindrome-partitioning/

Example:
Input: "aab"
Output:
[
  ["aa","b"],
  ["a","a","b"]
]

(Pending) Boolean Parenthesization

https://www.interviewbit.com/problems/evaluate-expression-to-true/

Example:

Let’s say our expression is “T|F&T^F”, we apply MCM here and k will break the expression at every operation, for example: (T)|(F&T^F), (T|F)&(T^F) etc.
Now let’s figure out what i and j should be initially. Since here we don't have to worry about (i-1), we can choose i=0 and j=n-1. Next we need to decide what the base condition should be. Here the base condition should be either true/false, or 0/1, because we don’t always have to return true since let's say we have an expression (T)^(F), for the right partition if we don't return false, our overall expression won’t be true; so we have to take care of both true and false.

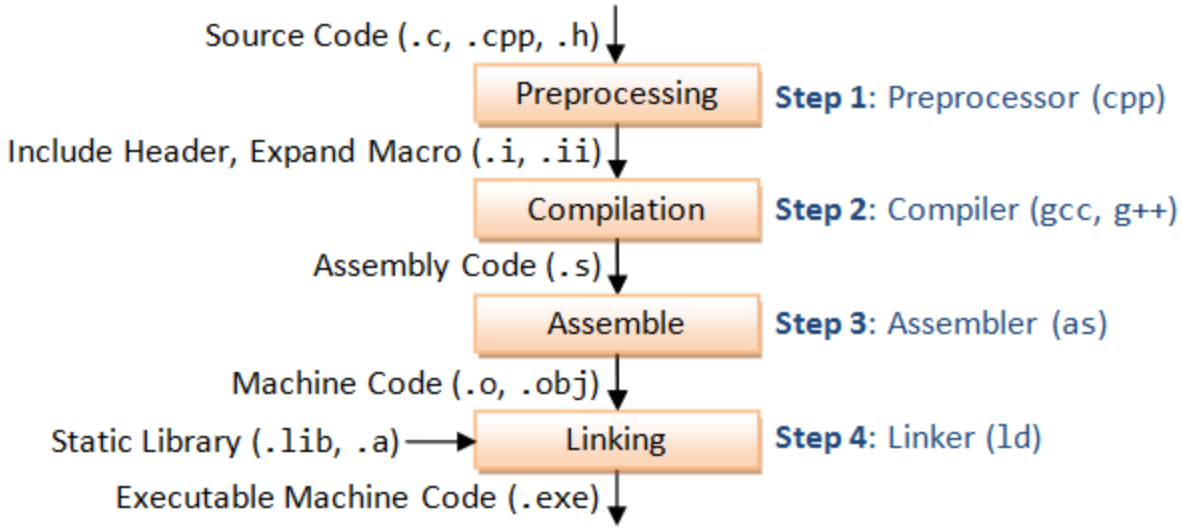
(Pending) Scrambled Strings

https://www.interviewbit.com/problems/evaluate-expression-to-true/

Example:

DAY 29 (Operating Systems)

Compiler, Loader, Linker



You make a piece of code and save the file (Source code), then

Preprocessing :- As the name suggests, it's not part of compilation. They instruct the compiler to do required pre-processing before the actual compilation. You can call this phase Text Substitution or interpreting special preprocessor directives denoted by #.

Compilation :- Compilation is a process in which a program written in one language gets translated into another targeted language. If there are some errors, the compiler will detect them and report it.

Assemble :- Assembly code gets translated into machine code. You can call assembler a special type of compiler.

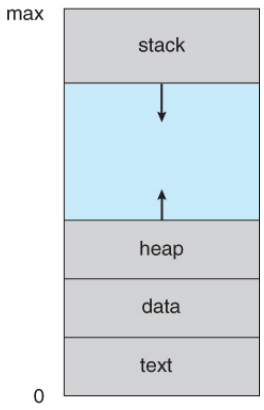
Linking:- If this piece of code needs some other source file to be linked, linker links them to make it an executable file. A linker is also responsible to link and combine all modules of a program if written separately.

Loader:- It loads the executable code into memory; program and data stack are created, register gets initialized.

What is a process?

A process is an instance of a program in execution. The process memory is divided into four sections primarily:

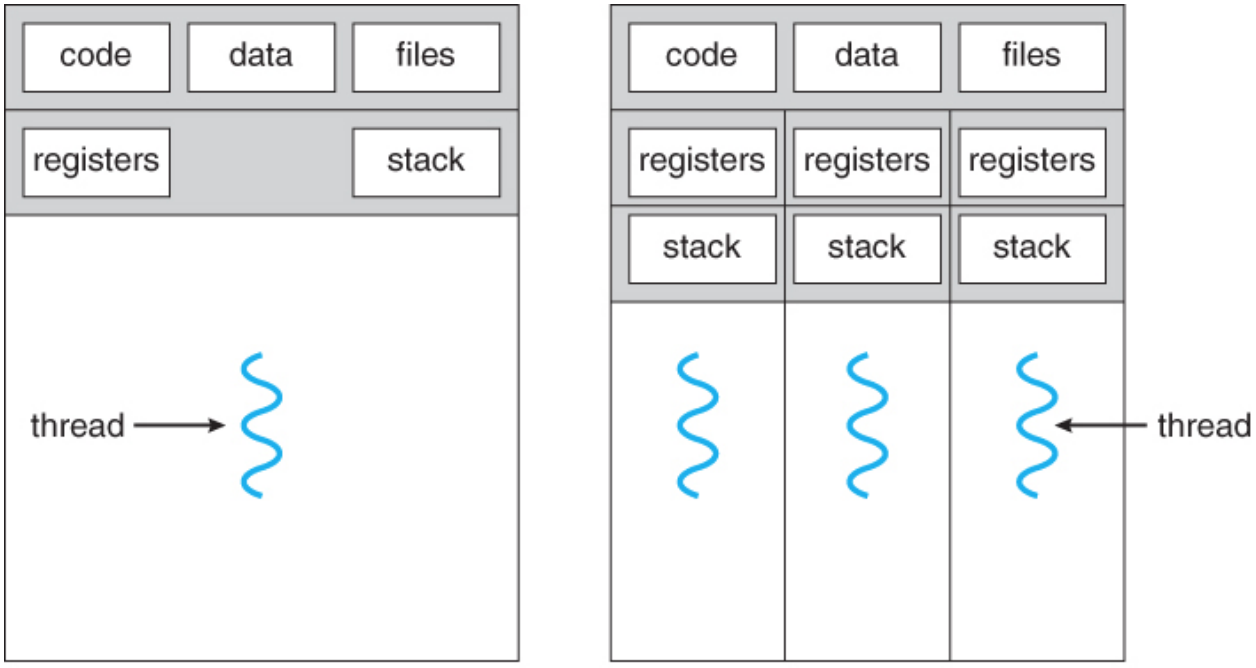
1. The text section comprises the compiled program code
2. The data section stores global and static variables, allocated and initialized prior to executing main
3. The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
4. The stack is used for local variables. Space on the stack is reserved for local variables when they are declared (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope.



Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.

What is a thread?

A ***thread*** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)

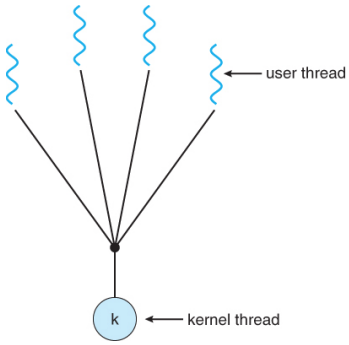


single-threaded process multithreaded process

Traditional processes have a single thread of control. Multi-threaded applications have multiple threads running in a process each having their own program counter, stack and registers, but they share the common code and data.

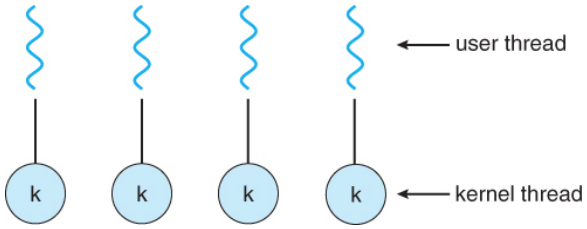
Multithreading models:

There are two types of thread to be managed: kernel threads and user threads. Kernel threads are supported within the kernel of the OS itself whereas user threads are the threads implemented by the programmers in their programs. For the working of the program, user threads must be mapped to the kernel threads, and one of the following models can be used.

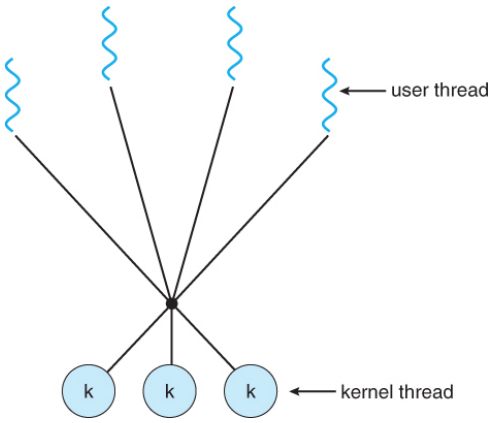


1. Many to one model: In this type of implementation all the user threads are mapped to one single kernel thread. Thread management is very efficient but, the kernel thread can map to only a single user thread at one point of time and hence if a blocking system call is made, then the

entire process blocks, even if the other user threads would otherwise be able to continue.



2. One to one model: If a blocking call is made the entire process doesn't get blocked and hence this resolves the problem in the previous model, but overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.



3. Many to many model: This gets the best of both the models mentioned above.The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads. Processes can be split across multiple processors.

Process Scheduling

Processes may be in one of 5 states:

- **New** - The process is in the stage of being created.
- **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
- **Running** - The CPU is working on this process's instructions.

- **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
- **Terminated -** The process has completed.

The act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.

The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

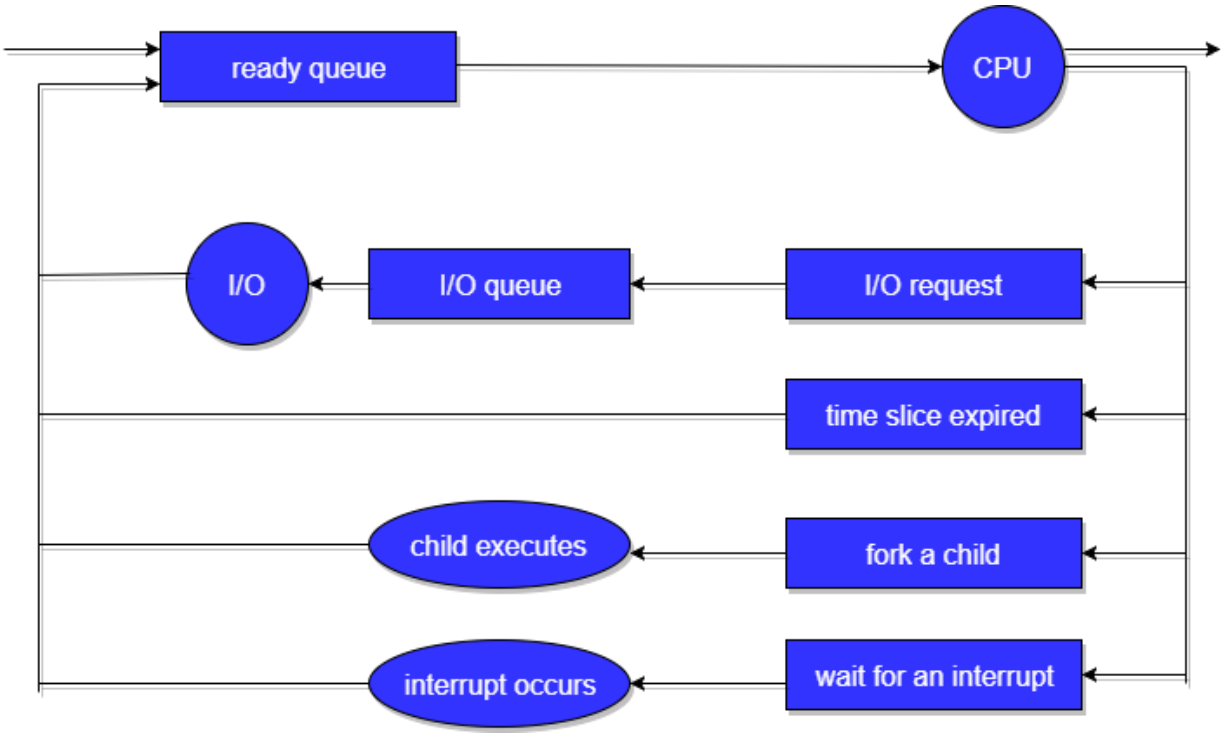
What are scheduling queues?

1. All processes, upon entering into the system, are stored in the **Job Queue**.
2. Processes in the Ready state are placed in the **Ready Queue**.
3. Processes waiting for a device to become available are placed in **Device Queues**. There are unique device queues available for each I/O device.

A new process is initially put in the **Ready queue**. It waits in the ready queue until it is selected for execution(or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

- The process could issue an I/O request, and then be placed in the **I/O queue**.
- The process could create a new subprocess and wait for its termination.

- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue



Types of schedulers

- A **long-term scheduler** is typical of a batch system or a very heavily loaded system. It runs infrequently, (such as when one process ends by selecting one more to be loaded in from disk in its place), and can afford to take the time to implement intelligent and advanced scheduling algorithms.
- The **short-term scheduler**, or CPU Scheduler, runs very frequently, on the order of 100 milliseconds, and must very quickly swap one process out of the CPU and swap in another one.
- Some systems also employ a **medium-term scheduler**. When system loads get high, this scheduler will swap one or more processes out of the ready queue system for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system.

Context Switching

Whenever an interrupt arrives, the CPU must do a **state-save** of the currently running process, then switch into kernel mode to handle the interrupt, and then do a **state-restore** of the interrupted process.

Similarly, a **context switch** occurs when the time slice for one process has expired and a new process is to be loaded from the ready queue. This will be instigated by a timer interrupt, which will then cause the current process's state to be saved and the new process's state to be restored.

Saving and restoring states involves saving and restoring all of the registers and program counter(s), as well as the process control blocks described above.

Context switching happens VERY VERY frequently, and the overhead of doing the switching is just lost CPU time, so context switches (state saves & restores) need to be as fast as possible.

Paging

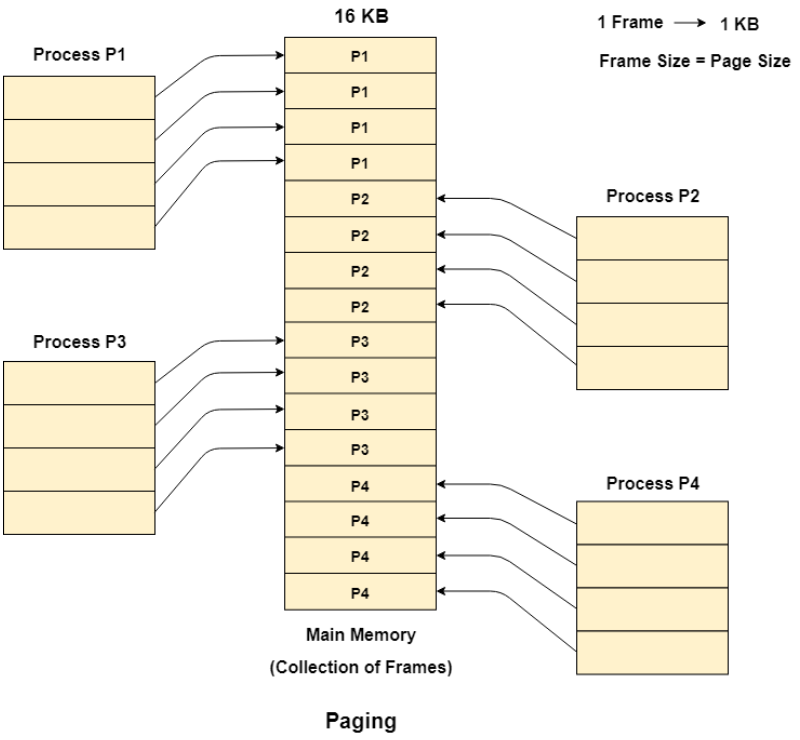
Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.

The main idea behind the paging is to divide each process in the form of pages. The main memory will also be divided in the form of frames.

One page of the process is to be stored in one of the frames of the memory. The pages can be stored at the different locations of the memory but the priority is always to find the contiguous frames or holes.

Let us consider the main memory size 16 Kb and Frame size is 1 KB therefore the main memory will be divided into the collection of 16 frames of 1 KB each.

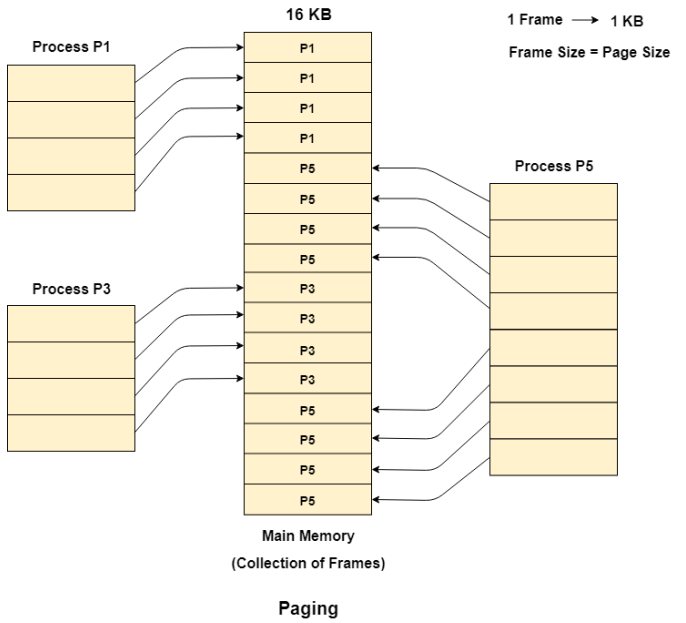
There are 4 processes in the system that are P1, P2, P3 and P4 of 4 KB each. Each process is divided into pages of 1 KB each so that one page can be stored in one frame.



Paging

Initially, all the frames are empty therefore pages of the processes will get stored in the contiguous way.

Let us consider that, P2 and P4 are moved to waiting state after some time. Now, 8 frames become empty and therefore other pages can be loaded in that empty place.



Paging

Segmentation

Segmentation is a memory management technique in which a process is divided into parts and then we put them into the main memory. Paging also does something similar, it divides the process into pages and then puts it into the main memory.

What is the difference? Let's say we have a function ADD(), now what paging does is that it simply divides it into equal sized pages, here assume it divides it into two pages, now these two pages are put into frames say f1 and f4. Now when CPU is executing, it first executes f1, but if you notice f1 doesn't contain the entire ADD() function, so because of this CPU only get's the partial code from f1. So simply put, paging divides all the process into the form of pages regardless of the fact that a process can have some relative parts of functions which need to be loaded in the same page.

Now, segmentation cares about the user's view of the process. It divides the process into the segments. Each segment contains the same type of functions such as the main function can be included in one segment and the library functions can be included in the other segment.

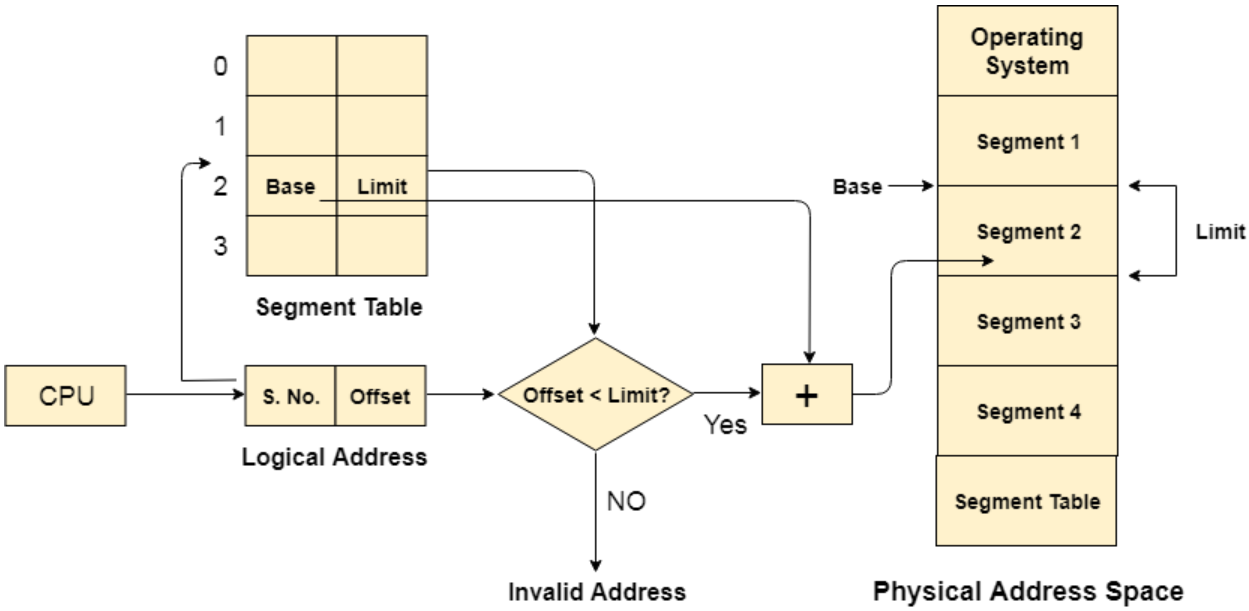
The details about each segment are stored in a table called a segment table. Segment table contains mainly two information about segment:

1. Base: It is the base address of the segment
2. Limit: It is the length of the segment.

CPU generates a logical address which contains two parts:

1. Segment Number
2. Offset/Segment Size

The Segment number is mapped to the segment table. The limit of the respective segment is compared with the offset. If the offset is less than the limit then the address is valid otherwise it throws an error as the address is invalid because offset just tells us what portion of the segment do we want to access, it can be smaller than the actual segment size or even equal to it, but not greater than it.



Semaphore vs Mutex

Go through this video to get a clear understanding of mutex, semaphores and different types of semaphores. https://youtu.be/8wcuLCvMmF8

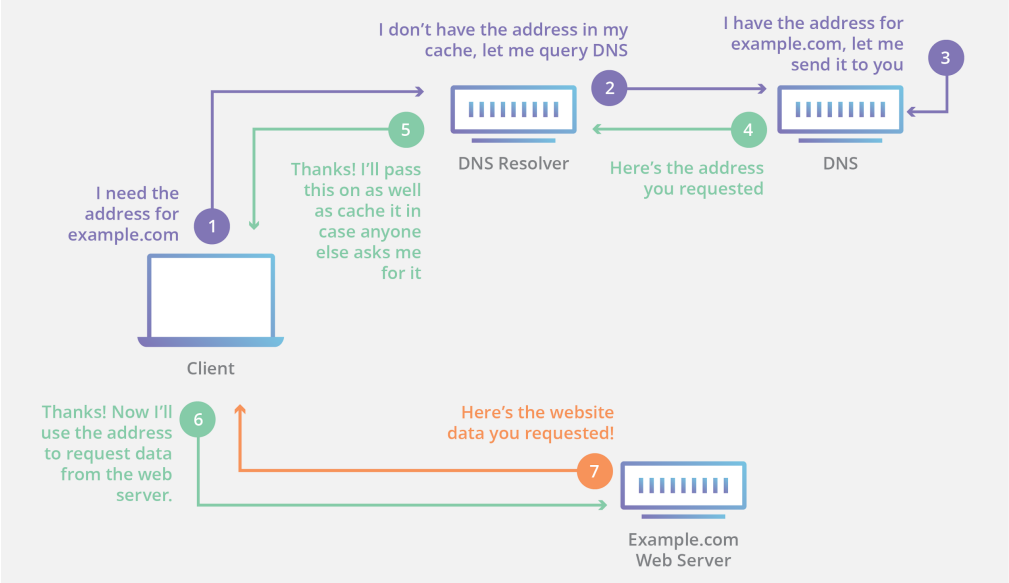
DAY 30 (Computer Networks)

DNS

The Domain Name System (DNS) is the phonebook of the Internet. Humans access information online through domain names, like nytimes.com or espn.com. Web browsers interact through Internet Protocol (IP) addresses. DNS translates domain names to IP addresses so browsers can load Internet resources. Each device connected to the Internet has a unique IP address which other machines use to find the device. DNS servers eliminate the need for humans to memorize IP addresses such as 192.168.1.1 (in IPv4), or more complex newer alphanumeric IP addresses such as 2400:cb00:2048:1::c629:d7a2 (in IPv6).

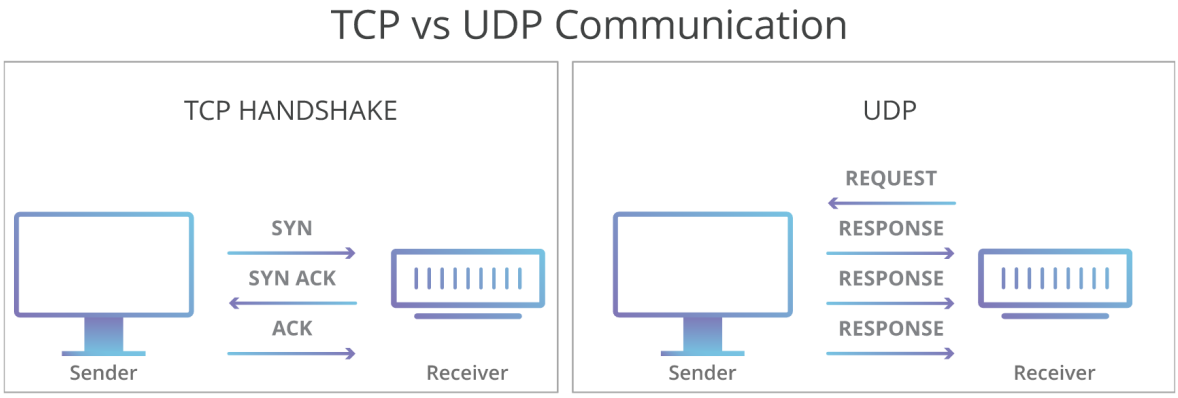
IP Address

The IP address tells us where you are and not who you are. MAC addresses tell us who you are as it is unique to each device and comes with your hardware. IP addresses are flexible and can change, whereas MAC addresses can't. Both IP address and MAC address are used to find where the particular message needs to be delivered.



UDP

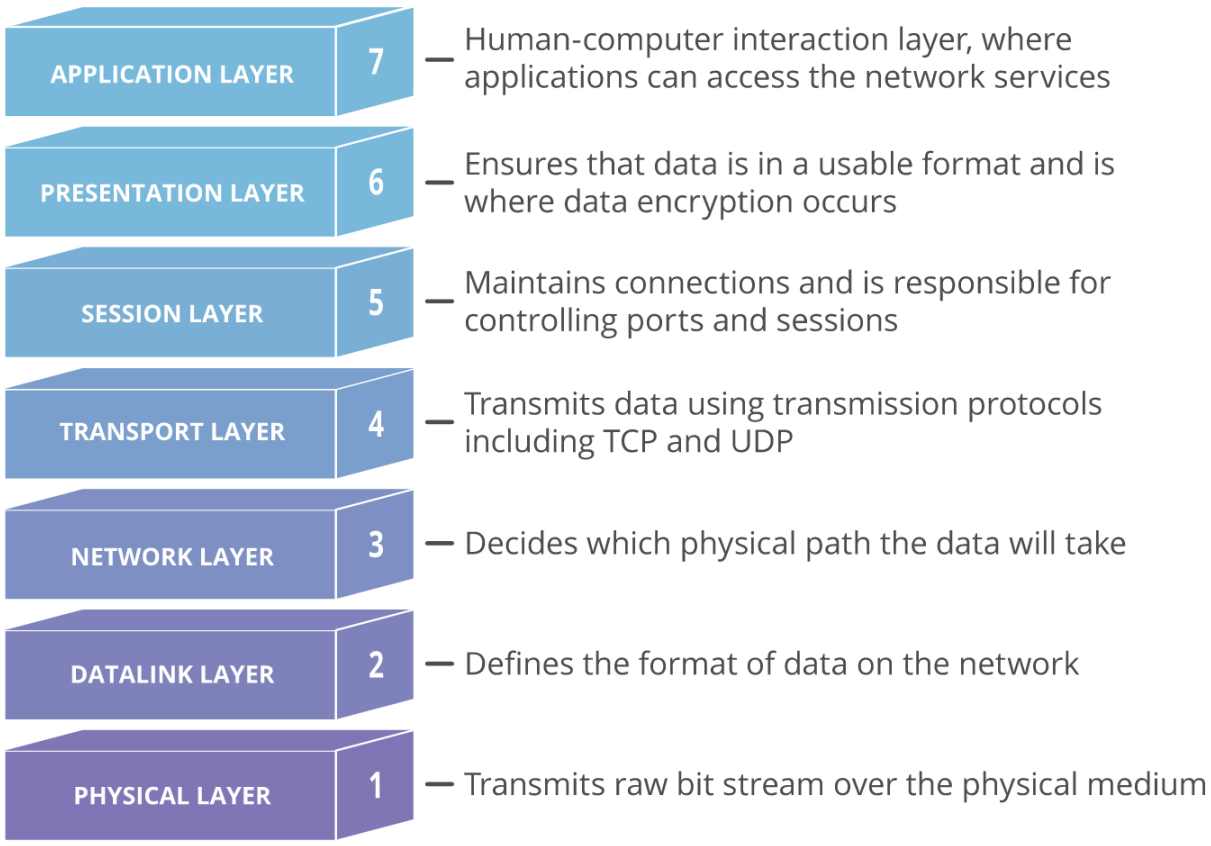
UDP is a communication protocol used across the Internet for especially time-sensitive transmissions such as video playback or DNS lookups. It speeds up communications by not requiring what's known as a "handshake", allowing data to be transferred before the receiving party agrees to the communication.



The OSI Model

https://www.cloudflare.com/en-gb/learning/ddos/glossary/open-systems-interconnection-model-osi/

OSI provides a standard for different computer systems to be able to communicate with each other. The OSI model can be seen as a universal language for computer networking. It's based on the concept of splitting up a communication system into seven abstract layers, each one stacked upon the last.



Watch this video for a detailed explanation of the OSI model along with TCP/IP model. https://youtu.be/vv4y_uOneC0

Subnetting/Supernetting

A subnet is a logical division of an IP Network. The process of dividing a network into two or more networks is called Subnetting. The main purpose is to relieve network congestion and improve network performance. Security is another benefit of subnetting.

Watch this video to understand how to subnet a network https://youtu.be/ecCuyq-Wprc

A subnet mask is used to determine if any two computers are on the same network or on different networks.

IPv4 vs IPv6

https://youtu.be/8npT9AALbrI