Arrays

Find largest element in array →

	TC	SC
Brute Force – Sort array and return last element	O(nlogn)	O(1)
Optimal – Traverse whole array, and compare with a variable "largest".	O(n)	O(1)

Find 2nd largest element in array →

	TC	SC
Brute Force – Sort array, and traverse from last element to find 2nd largest element	NlogN+N	O(1)
Better – Find largest in first iteration. Then find 2nd largest in another iteration	N+N	O(1)
Optimal –	N	O(1)

```
largest = arr[0], slargest = INT_MIN
if (arr[i] > largest) {
    slargest = largest;
    largest = arr[i];
}
else if (arr[i] < largest && arr[i] > slargest)
    slargest = arr[i];
```

Check if array is sorted →

	TC	SC
Optimal – Traverse whole array & compare with prev element.	O(N)	O(1)

Remove duplicates in place from Sorted array →

	TC	SC
Brute Force – Use a set data structure, & insert all elements in set. Inserting in a set takes O(NlogN)	O(N) + O(NlogN)	O(N)
Optimal – arr[] = {1, 1, 2, 2, 2, 3, 3} (i, j)	O(N)	O(1)

Two pointer approach – We know array is sorted
~~at~~ first, we put arr[0]. Then we find arr[i] != arr[j]
Then we put arr[j]. Then i becomes j.
We repeat the above process

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

Left Rotate the array by one place →

Traverse whole array, & put arr[i] = arr[i+1];
After loop arr[n-1] = temp;
where temp was arr[0];

TC	SC
O(N)	O(1)

Left Rotate by k places →

Brute Force – Take a temp array.
Store k to n elements in temp. Then store 0 to k elements in temp,
Copy elements of temp in arr,
3 loops

SC	TC
O(N)	O(3N)

Optimal – ~~Rotate~~ Reverse 0 to k elements in array
Reverse k to d elements in array
Then reverse whole array.

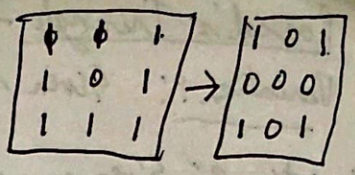
TC

reverse(arr, arr+d); O(d)
reverse(arr+d, arr+n); O(n-d)
reverse(arr, arr+n); O(n)

→ O(d+n-d+n)
O(2n)

Move Zeroes to End →

Set Matrix Zero → *



Declare two arrays: a row array of size N
col array of size M

Initialized with 0

If any cell(i,j) contains 0, we will mark ith index of row array & jth index of col array as 1.

In second time wherever row[i] || col[j] == 1, then we put zero in that location in matrix

O(M*N) — TC O(M) + O(N) — SC

Optimal → Use first row & first col of matrix as row & col array. No need to make separate arrays.

For mat[0][0], if both rows and cols need to be changed, then we will use a variable col0, and set it to 0.

In first loop we will make all 1 to n-1 cols & rows 0, wherever needed, then in second loop we will traverse only first row and first col, and check whether any variable needs to be changed

Note: First we will change row first, then the column.

T.C - O(N*M)

S.C O(1)

Zeller's Congruence algo →

To find day for any date

For Gregorian calendar

$$h = \left[q + \left[\frac{13(m+1)}{5}\right] + K + \frac{K}{4} + \frac{J}{4} + 5J\right] \bmod 7$$

h → day of week (0 = Saturday, 1 = Sunday .... 6 = Friday)

q → day of month

m → month

k → year % 100 (year of the century)

J → year / 100 (zero-based century. e.g. - 1995 → 19th century)

```
if (m < 3) {
    m = m + 12
    year--;
}
```

Pascal's Triangle →

Variation 1: Given row r & column c. Print element at position (r, c)

Variation 2: Given row n. Print n^{th} row of Δ.

Variation 3: Given n rows. Print n rows of Δ.

In Pascal's Δ, each number is sum of above 2 numbers

Variation 1: $^{r-1}C_{c-1}$ will be the element we need.

$^nC_r = \frac{n!}{(n-r)!\,r!}$

Naive approach- Calculate n!, r!, (n-r)! separately then calculate ^nC_r.

T.C - $O(n) + O(r) + O(n-r)$

Optimal approach-

optimized formula

$$^nC_r = \frac{n \times n-1 \times (n-2) \ldots\ldots (n-r+1)}{r \times r-1 \times r-2 \ldots\ldots 1}$$

So, $\frac{n}{1} \times \frac{n-1}{2} \times \ldots\ldots \frac{n-r+1}{r}$

Code-

```
long long res = 1;
for (i = 0 to r)
{
res = res * (n-i);
res = res / (i+1);
}
```

Variation 2:

Same code as above. We will call the function in loop again & again.

```
for (int c = 1; c <= n; c++){
    cout << nCr(n-1, c-1) << " ";
}
```

Variation 3:

```
curr = prev * (row - col) / col;
```

Approach: 1. Print 1 manually
2. Use a loop from 1 to n-1, and print rest

```
ans = ans * (n-i);
ans = ans / i;
cout << ans << " ";
```

Next Permutation →

Next lexicographical permutation.

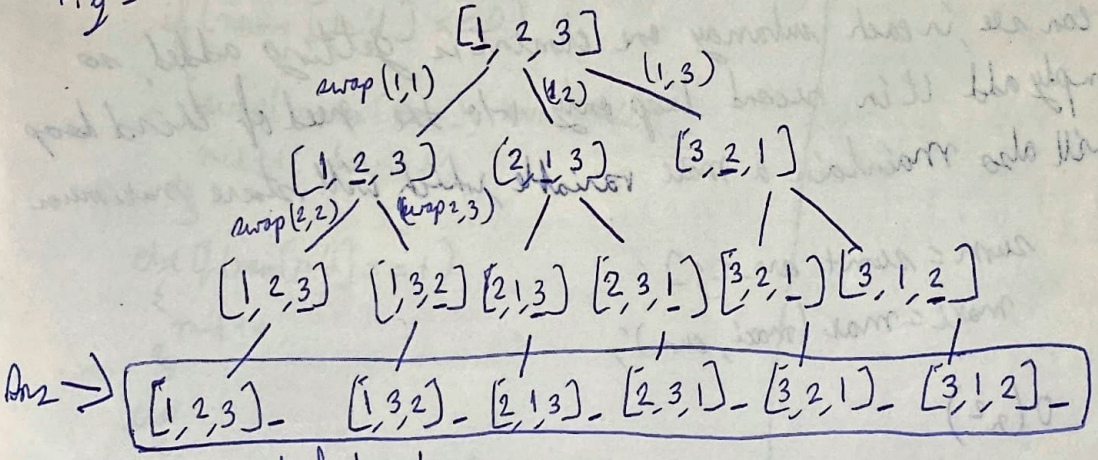
Brute force - Find all possible permutations, and then find next permutation

```
void permute(int index, vector nums, vector of vector ans)
{
    if (index >= nums.size())
    {
        ans.push_back(nums);
        return;
    }

    for (int i=index; i<nums.size(); i++)
    {
        swap(nums[i], nums[index]);
        permute(index+1, nums, ans);
        swap(nums[index], nums[i]);
    }
}
```

Approach: Here approach is to swap each index with every other index and then print permutations.

e.g. -



Ans → [1,2,3] [1,3,2] [2,1,3] [2,3,1] [3,2,1] [3,1,2]

When index goes out of bounds.

Optimal -

1. Find break-point, (break point means first index i from back of the array where arr[i] < arr[i+1]).

   E.g. - {2, 1, 5, 4, 3, 0, 0}, break point is index = 1.

2. If break point does not exist, array is sorted in decreasing order, the given permutation is last one in sorted order. So, next permutation will be the first.
   ∴ We will reverse the array and return it.

3. If break-point exists,
   - Find smallest number i.e. > arr[i] and in the right half of breakpoint and swap it with arr[i].
   - Reverse entire right half & return the arr

Maximum Subarray Sum; (Kadane's algo)

E.g - [-2, 1, -3, 4, -1, 2, 1, -5, 4]

Subarray with max sum is [4, -1, 2, 1]

Brute force-

Check sum of every possible subarray and consider max among them. Using 3 nested loops. First loop (i, j) will iterate over every possible starting index & ending index.

Subarray → index i to index j. Using another loop we will get the sum of the elements of the subarray [i.....j].

T.C- $O(n^3)$

Better approach-

Using two loops

We can see, in each subarray one element is getting added, so we will simply add it in second loop only w/o the need of third loop and we will also maintain a maxi variable, which will store maximum sum,

```
sum = sum + arr[j];
maxi = max(maxi, sum);
```

T.C- $O(n^2)$

Optimal approach - (Kadane's algo)

Intuition of this algo is that to consider the subarray as a part of the answer if its sum < 0.

- Iterate arr with single loop, while iterating add elements in sum variable. Whenever sum < 0, make sum = 0. Along with sums calculated we will consider max one.

T.C- $O(n)$

```
sum += arr[i];
maxi = max(sum, maxi);
if (sum < 0) sum = 0;
```

Sort an array of 0s, 1s & 2s →

Brute Force → Use merge sort / any sort
TC - O(n log n)

Better → Count 0s, 1s & 2s in one loop.
Then in second loop, we will put values accordingly.
First 0s, then 1s & then 2s

Optimal → Use 3 variables low = 0, mid = 0, high = n-1;

- Intitution is → low to mid-1 is sorted
mid to high is unsorted
high to n-1 is sorted.

```
while(mid <= high)
{
  if(nums[mid] == 0)
  {
    swap(nums[low], nums[mid]);
    mid++; low++;
  }
else if(nums[mid] == 1)
{
  mid++;
}
else
{
  swap(nums[mid], nums[high]);
  high--;
}
}
```

This will sort the entire array.

Stock Buy and Sell →

Find max profit. [7, 1, 5, 3, 6, 4] O/P → 5

Brute Force: Use 2 loops and track every transaction. O(n²)

Optimal: Linearly traverse the array.
Maintain a min from start of array & compare with every element,
if it is greater than min, then take the diff & maintain it in max,
otherwise update min.

```
min = arr[0];
max = INT_MIN;
for(i=1 to n) {
  int diff = arr[i] - min;
  max = max(diff, max);
  min = min(prices[i], min);
```

Rotate Matrix by 90 deg →

1 2 3	7 4 1
4 5 6	8 5 2
7 8 9	9 6 3

Approach - Make a dummy matrix of n*n, take first row of matrix and put in last col of dummy matrix, and so on.

rotated[j][n-i-1] = matrix[i][j];

Optimised - Transpose the matrix, then reverse each row

For transpose → swap(matrix[i][j], matrix[j][i]);

For reverse → reverse(matrix[i].begin(), matrix[i].end());

In transpose, we only need to swap upper Δ & lower Δ elements, i.e. non diagonal elements. So, second loop will run from 0 to i

Merge Overlapping Sub Intervals → *

I/P → [[1,3],[2,6],[8,10],[15,18]] O/P → [[1,6],[8,10],[15,18]]

Brute Force

Select one interval and compare it with all remaining intervals using 2 loops and store it in different one array.

The array must be sorted before this.

1. How to check whether interval can be merged with selected interval →

Compare current interval's start with selected interval's end

```
if(start <= selected[i][1])
{
    end = max(end, selected[j][1]);
}
```

If can be merged, then update end of last inserted interval with max(current interval's end, selected interval's end)

2. If cannot be merged, then insert as it is.

We will also skip those intervals which have been merged, using this →

```
if(ans.size() && end <= ans[n-1][1])
    continue;
```

S.C - O(2n) T.C - O(NlogN) + O(N)

Optimal - Sort the array

We will push whatever interval that comes into the ans array. Before pushing we will compare whether its start is less than the end of interval in ans array. If yes, then update the interval in ans array by comparing ans interval's end with ans interval's end, whichever is max will replace it.

This all will be done in one loop.

T.C - O(n) S.C - O(2n)

Merge Two Sorted Arrays W/O Extra Space *

Brute Force: Using extra space O(m+n).

Compare elements of both arrays and insert in new array accordingly.

Optimal 1: left = m-1 to 0, right = 0 to n where m = arr1.size;
n = arr2.size;

```
if (arr1[left] > arr2[right])
{swap( )
 left--, right++;
}
else
{
break;          // All elements before left & after right
}                  will be sorted.
```

Then later, we sort these 2 arrays individually and then merge.

In the end, the left array (arr1) will contain small values & arr2 contains large values.

TC → O(max(m,n) + O(n log n) + O(m log m)

SC → O(1)

Optional 2 → Using Gap method, (comes from Shell sort)

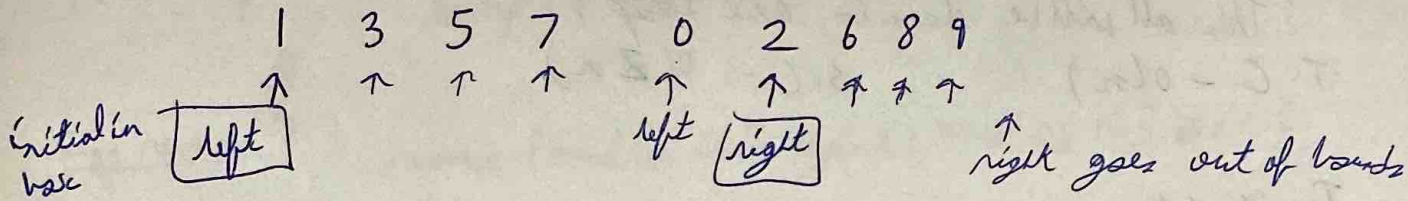
arr1[] = {1, 3, 5, 7} arr2[] = {0, 2, 6, 8, 9}

n = 4 m = 5

$\frac{4+5}{2} = \lceil 4.5 \rceil$ = ceil of (4.5) = 5

So, Initially gap = 5

First pointer at i = 0 second pointer at i+gap away

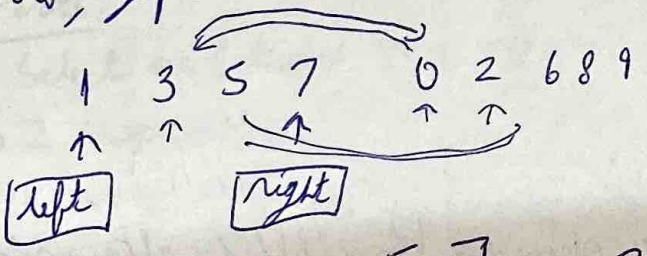


Start comparing, left & right.

Whenever gap goes out of bounds, we restart by reducing the value of gap in next iteration.

$gap = 5 = \frac{5}{2} = 2.5 = ceil[2.5] = 3$

Now, gap = 3



Now $gap = \frac{3}{2} = \lceil 1.5 \rceil = 2$

Then $gap = 1 = \frac{1}{2} = \lceil 0.5 \rceil$; again gap becomes 1,

gap = len/2 + len % 2

T.C – O(n+m * log(n+m))

Find the duplicate in an array of N+1 integers →

arr = [1, 3, 4, 2, 2] O/P = 2

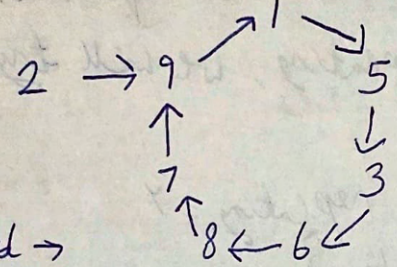
Brute Force- Sort and then compare if arr[i] = arr[i+1]

Better- Use map or use count array

Optimal- Uses Linked List cycle method,

0	1	2	3	4	5	6	7	8	9
2	5	9	6	9	3	8	9	7	1

Initially index=0, as arr[0]=2, then we go to
index=2, arr[2]=9, then we go to arr[9]
index=9, arr[9]=1,



After creating cycle, we will use Tortoise method

Tortoise method →

One is slow pointer, another is fast pointer
slow pointer moves one, fast moves 2
Wherever slow & fast collides, we will put fast again in first pointer. Then we move both fast & slow by one point.
The point where they collide for second time, will be the answer.

T.C - O(N) S.C - O(1)

```
slow = nums[0]
fast = nums[0]

do{
  slow = nums[slow]
  fast = nums[nums[fast]]
  }while(fast != slow);
fast = nums[0];
while(slow != fast)
  {
  slow = nums[slow];
  fast = nums[fast];}   return slow;
```

Find the repeating and missing number → *

{3, 1, 2, 5, 3}

Ans → {3, 4}

3 is repeating
4 is missing

Brute Force: Use two loops, outer loop will run from 1 to n and compare each number with each element in the array which will be iterated by inner loop. Maintain a cnt variable in each iteration, if (cnt == 2) repeating ; if (cnt == 0) (missing);

$O(n^2)$ - TC $O(1)$ - SC

Better Approach: (Using Hashing) TC - $O(n)$ | SC - $O(n)$

Optimal Approach 1: (Using maths)

Convert the given problem into mathematical equations.

We have 2 variables, i.e. missing and repeating, we will try to form two linear equations.

Assume, missing = x & repeating = y

Sum of first N nos. → $S_n = \frac{N(N+1)}{2}$

S = Summation of all elements in the given array

$S - S_n = X - Y$ — (1)

Sum of first N squares → $S2_n = \frac{N(N+1)(2n+1)}{6}$

S2 = Summation of all squares of elements in given array

$S2 - S2_n = X^2 - Y^2$ — (2)

TC - $O(n)$ SC - $O(1)$

~~Approach 2:~~

Count inversions in an array → (Using merge-sort)

Inversion: for all $i, j <$ size, if $i < j$ then find pair $(A[i], A[j])$ s.t $A[j] < A[i]$.

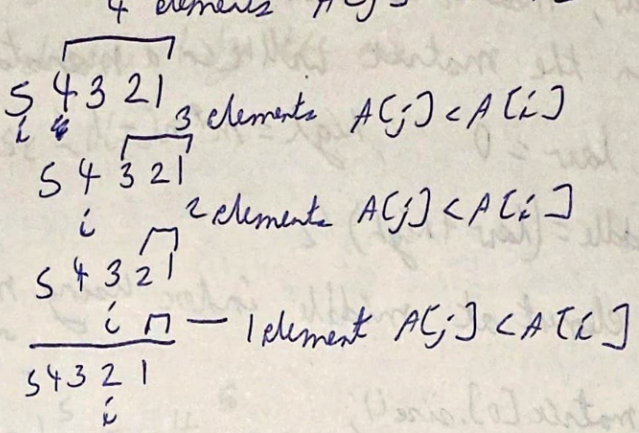
E.g- N = 5 arr[] = {1, 2, 3, 4, 5}

O/P → 0

E.g- N = 5 arr[] = {5, 4, 3, 2, 1}

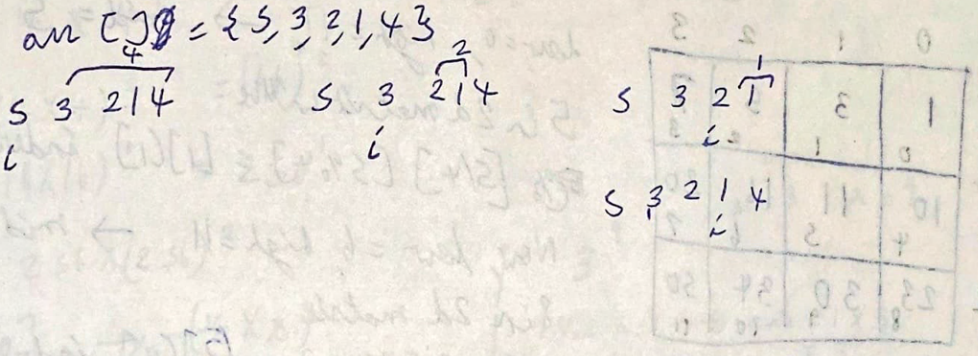
O/P → 10

Explanation →



∴ 4+3+2+1 = 10

E.g- N = 5 arr[] = {5, 3, 2, 1, 4}



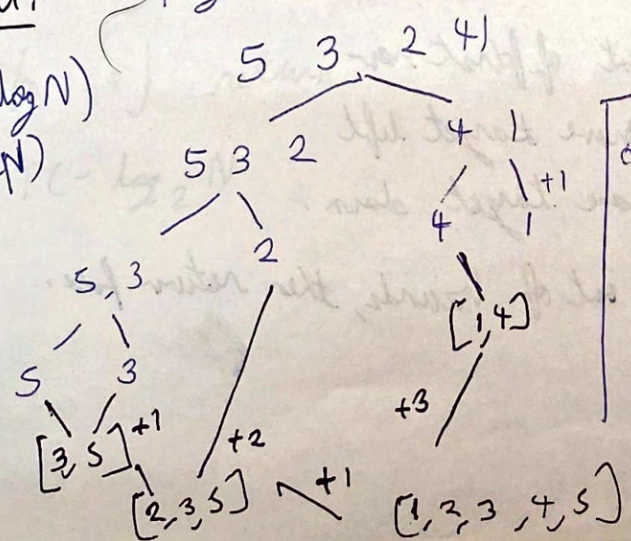
4+2+1 = 7

O/P → 7

Brute force: Using two loops and counting arr[j] < arr[i] for every index i. TC - $O(n^2)$ SC - $O(1)$

Optimal: Merge Sort

TC - O(N log N)
SC - O(N)



cnt → +1 +2 +1 +3 +1

Add this line

```
cnt += mid - left + 1;
```

Where arr[j] > arr[i] in merge function of merge sort

Search in a sorted 2D matrix →

$$\begin{bmatrix} 1 & 3 & 5 & 7 \\ 10 & 11 & 16 & 20 \\ 23 & 30 & 34 & 60 \end{bmatrix}$$

target = 3

O/P → true

Naive approach: Traverse through every element and search

TC - O(m*n) SC - O(1)

Optimal - (Binary Search)

Since know, matrix is row-wise and column-wise sorted the elements in the matrix will be in a monotonically increasing order.

Initially, low = 0, high = m*n - 1;

middle = (low + high) / 2

We can get element at middle index using mat[middle/m][middle%m]

```
m = matrix[0].size();
n = matrix.size();
```

	0	1	2	3
0	1 (0)	3 (1)	5 (2)	7 (3)
1	10 (4)	11 (5)	16 (6)	20 (7)
2	23 (8)	30 (9)	34 (10)	50 (11)

low = 0, high = 11 → mid = 5

5 in 2d matrix will be [5/4][5%4] = [1][1] index

Now, low = 6, high = 11 → mid = 8

8 in 2d matrix,

[8/4][8%4] = [2][0] index

TC → $\log_2(m \times n)$

SC → O(1)

Better Optimal - start from last element of first row

if (ele < target), move target left

ele > target, move target down.

If any i or j goes out of bounds, then return false.

Pow(x, n) →

Given a double x and integer n, calculate x^n.

Basically implement pow(x, n).

E.g:- x = 2.00000 , n = 10

O/P → 1024.00000

E.g:- x = 2.10000 , n = 3

O/P → 9.26100

Brute Force →

Looping from 1 to n, and keeping a ans (double) variable

Every time multiply x with ans.

If n is negative, divide 1 by the ans

TC - O(N) SC - O(1)

Optimal →

Binary Exponentiation

$2^{10} = (2 \times 2)^5 = 4^5$

$4^5 = 4 \times 4^4 = 4 \times 256 = 1024$

$4^4 = (4 \times 4)^2 = (16)^2$

$16^2 = (16 \times 16)^1 = 256^1$

$256^1 = 256 \times (256)^0$

[n % 2 == 0] → (x × x), (n / 2)

[n % 2 == 1] → ans = ans × x, n = n - 1

if (n == 0) return

TC - $\log_2 N$

ans = 1

x = 3 n = 9

3^9

9 % 2 == 1 → ans = 1 × 3 = 3, n = 9 - 1 = 8

8 % 2 == 0 → x = 3 × 3 = 9, n = 8/2 = 4

4 % 2 == 0 → x = 9 × 9 = 81, n = 4/2 = 2

2 % 2 == 0 → x = 81 × 81 = 6561, n = 2/2 = 1

1 % 2 == 1 → ans = 3 × 6561 = 19683, n = 1 - 1 = 0

(n == 0) return

Majority Element > N/2 times →

N = 3 nums = {3, 2, 3}

Ans - 3

N/2 = 1

Count of 3 = 2 ← Ans

Count of 2 = 1

N = 7 nums = {2, 2, 1, 1, 1, 2, 2}

7/2 = 3

O/P → 2

Naive approach: Run 2 loops to find occurence of each element and find those whose count > N/2.

TC - O(N²)

Better: Hashing TC → O(N log N) + O(N)
SC → O(N)

Optimal: [Moore's Voting Algorithm]

1 → Apply Moore's algo

2 → Verify answer by iterating over the array again

	0	1	2	3	4	5		6	7		8	9	10	11		12	13	14	15	
arr[] = {	7	7	5	7	5	1		5	7		5	5	7	7		5	5	5	5	]
	cnt = 0							cnt = 0			cnt = 0					cnt = 4				

Initially,
cnt = 0, ele = ?,

We start iterating, We are assuming arr[0] is the answer, so we put it in ele.

i = 0
cnt = 1, ele = 7

at i = 1
cnt = 2, ele = 7

at i = 2, whenever arr[i] != ele, it decrements
cnt = 1, ele = 7
cnt = 2, ele = 7
cnt = 1, ele = 7
cnt = 0, ele = 7

[0 - 5]

Now, cnt again became 0, we can say that till this part of the array there is no majority element, as cnt became 0.

Similarly, we will iterate further, now ele will reset

ele = 5, cnt = 1

, cnt = 0, it again become 0 by next element.

So, no majority element in this section. [6-7]

Again,

ele = 5, cnt = 1

cnt = 2

cnt = 1

cnt = 0, No majority element in this section [8-11]

Now, in [12-15]

ele = 5, cnt = 4, this is the majority element.

Now, we will exit the loop and recheck ele, whether it is majority element or not. If it doesn't return -1

TC - O(N) SC - O(1)

Majority Element > N/3 times →

N = 5, arr[] = {1, 2, 2, 3, 2}

N/3 = 1

O/P → 2

N = 6, arr[] = {11, 33, 33, 11, 33, 11}

O/P → 11 33

At max, there will just be 2 elements

Because, i.e. $\left\lfloor \frac{9}{3} \right\rfloor = 3$

$> 3 = 4$

An array of size 9 can have 4 + 4 + 1 = 9 elements

So, at max only 2 elements can have freq > 3

Brute Force → Using 2 loops and count occurrence of each element.

We can save some iterations by checking whether that element is present or not in the list.

The end point will be when the ans vector / list has 2 elements in it.

```
if (ls.size()==0 || ls[0] != nums[i])
  {
    cnt = 0;
    for(j = 0 to n-1)
        if (nums[j]==nums[i])
          cnt++;

    if (cnt > n/3)
        ls.add(nums[i]);
  }
```

TC - $O(n^2)$
SC - $O(1)$

Better - Using Hashing — SC - $O(n)$, TC - $O(n \log n)$

Optimal - Moore's voting algo.

```
cnt1 = 0,  cnt2 = 0
el1        el2

if (cnt1==0 && nums[i] != el2)
  {
  }
else if (cnt2==0 && nums[i] != el1)
  {
  }
else if (el1 == nums[i]) cnt1++;
else if (el2 == nums[i]) cnt2++;
else
  { cnt1--, cnt2--;
  }
```

TC - $O(N)$
SC - $O(1)$

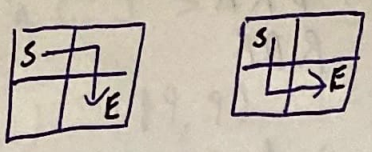
After this manually check count of el1 & el2 by traversing whole array again. if cnt1 > N/3 then push el1 & cnt2 > N/3 then push el2

Grid Unique Paths →

Count paths from left-top to the right bottom of a matrix.

E.g- m=2, n=2

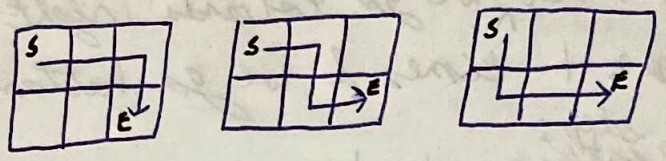
O/P → 2



We can only move right & bottom direction

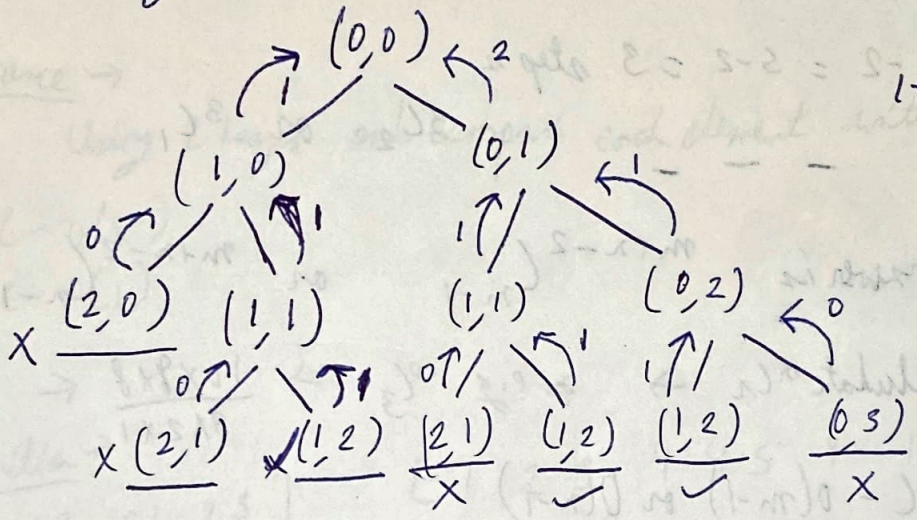
E.g- m=2, n=3

O/P → 3



Brute Force → Using recursion, try all possible combinations

E.g-2




```
int solve(int i, int j, int m, int n)
{
  if(i>=m || j>=n) return 0;
  if(i==m-1 && j==n-1) return 1;
  return solve(i+1,j,m,n) + solve(i,j+1,m,n);
}
```

TC- Exponential $O(2^n)$

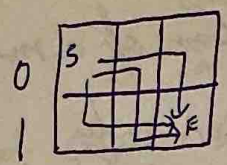
SC- Exponential $O(2^n)$ because of recursive stack

Better → Add dp to this soln

TC- $O(n*m)$

SC- $O(n*m)$

Optimal → Using Combinations



3 ways.

RRD
DRR
RDR

m=2, n=3

Obs 1: We always take 3 steps to reach End from Start

Observation 2: To reach end we go towards right 2 times, i.e., n-1 times & we go bottom 1 time, i.e., m-1 times

Total steps = m-1 + n-1
= m+n-2

Here, 2+3-2 = 5-2 = 3 steps

$^3C_2$ or $^3C_1$

- - -

∴ Our answer is $^{m+n-2}C_{m-1}$ or $^{m+n-2}C_{n-1}$

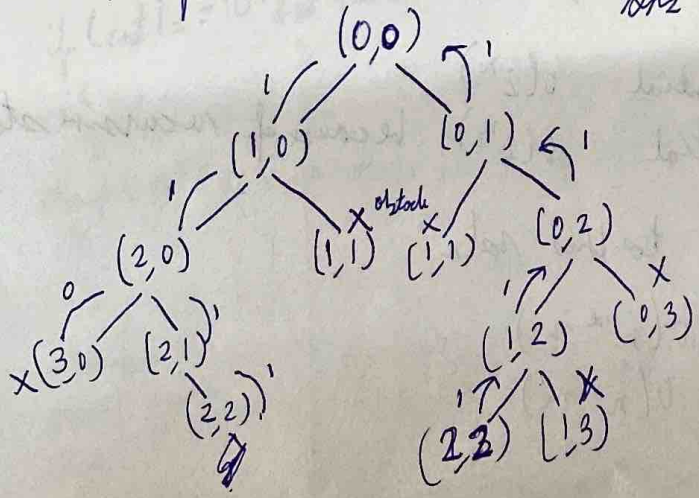
Shortcut to calculate ^nC_r → e.g. $^{10}C_3$ → $\frac{10 \times 9 \times 8}{3 \times 2 \times 1}$

TC - O(m-1) or O(n-1)
SC - O(1)

Unique Paths II - (Same ques with obstacle in b/w paths)
Calculate possible paths / no. of ways
1 represents an obstacle.

0 0 0
0 1 0
0 0 0

Ans - 2



Reverse Pairs →

arr[] = {40, 25, 19, 12, 9, 6, 2}

i < j && a[i] > 2 X a[j]

(6, 2) (19, 6) (19, 9) (25, 12) (40, 19)
(9, 2) (25, 6) (25, 9) (40, 12)
(12, 2) (40, 6) (40, 9)
(19, 2)
(25, 2)
(40, 2)

6 + 3 + 3 + 2 + 1 = 15

Brute Force →

Using 2 loops and compare each element with other

TC - O(n²)
SC - O(1)

Optimal →

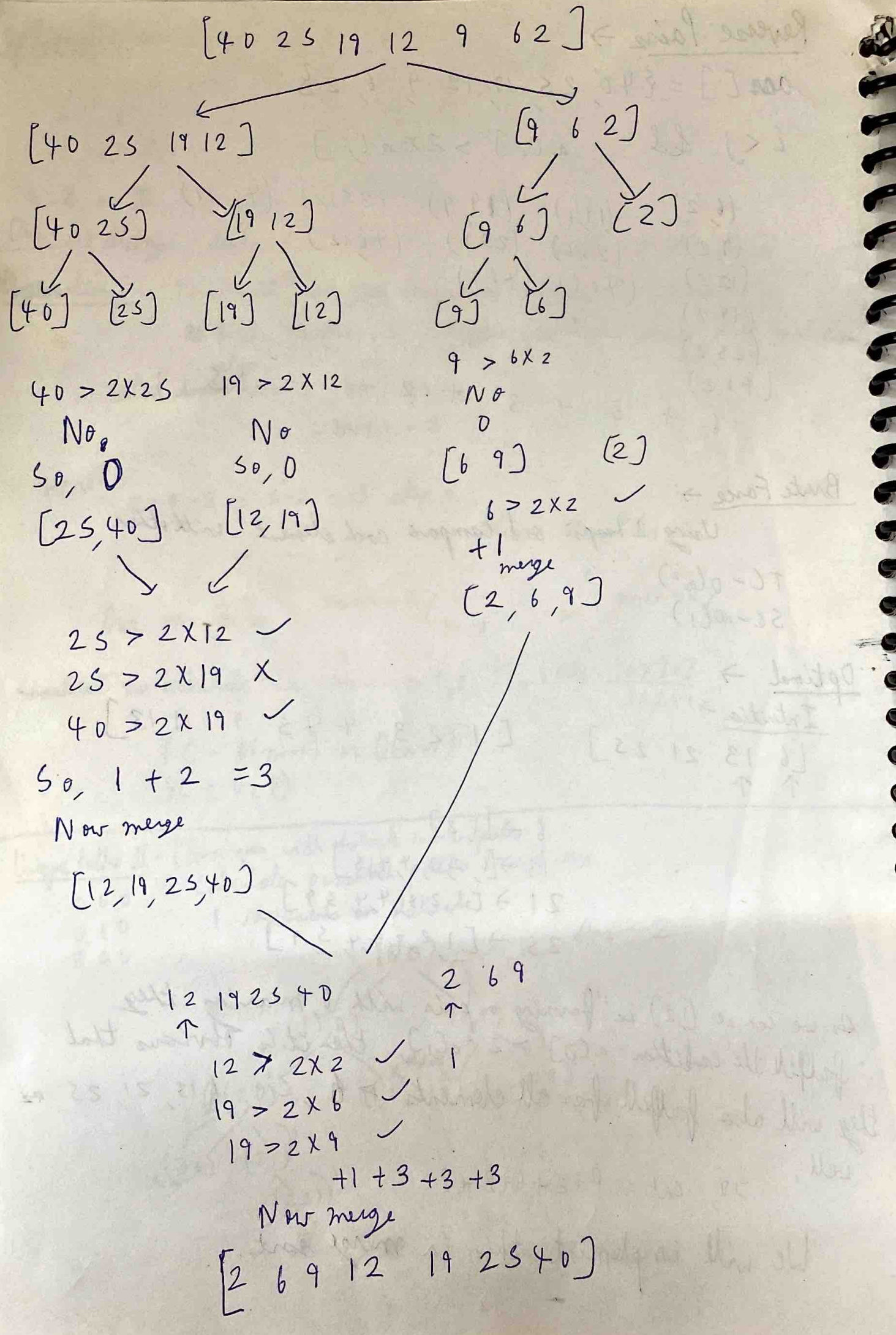
Intuition →

[6 13 21 25] [1 2 3 4 4 5 9 11 13]
↑ ↑

6 → [1, 2]
13 → [1, 2, 3, 4, 4, 5]
21 → [1, 2, 3, 4, 4, 5, 9]
25 → [1, 2, 3, 4, 4, 5, 9]

As, we can see (1,2) is forming a pair with 6, meaning they fulfill the condition a[i] > 2 * a[j], then it is obvious that they will also fulfill for all elements > 6, i.e. 13, 21, 25 as well. So, cnt = 0+2+6+7+8

We will implement this in merge sort.


[40 25 19 12 9 6 2]
[40 25 19 12]
[9 6 2]
[40 25]
[19 12]
[9 6]
[2]
[40]
[25]
[19]
[12]
[9]
[6]
40 > 2X25
No
So, 0
19 > 2X12
No
So, 0
9 > 6X2
No
0
[25, 40]
[12, 19]
[6 9]
(2)
6 > 2X2
+1
merge
[2, 6, 9]
25 > 2X12
25 > 2X19 X
40 > 2X19
So, 1 + 2 = 3
Now merge
[12, 19, 25, 40]
12 19 25 40
2 6 9
12 > 2X2
19 > 2X6
19 > 2X9
1
+1 +3 +3 +3
Now merge
[2 6 9 12 19 25 40]

We will have to add some code in b/w before merging to compare, and count reverse pairs

```
cnt = 0,   right = mid+1
for (i = low → mid)
{
  while ( right <= high && a[i] > 2*a[right])
    right++;
  cnt = cnt + (right - (mid+1))
}
```

TC - $\log n \times (n+n)$

SC - $O(n)$

Two Sum:

N = 5 arr[] = {2, 6, 5, 8, 11} target = 14

O/P → true [6, 8]

Brute force → For each element, we will try to search for another element such that its sum == target.

TC - $O(N^2)$ SC - $O(1)$

Better → (Using hashing)

Instead of using a loop to search for another element, we will use Hashmap to check whether target - (selected element) exists.

x → currently selected element [2, 3, 1, 4] target = 4

For i = 0, x = 2 and map is empty

We find, 4 - 2 = 2 in map, but map is empty right now, so store index of 2 in map. i.e. mp[2] = 0

i = 1, x = 3

Find 4 - 3 = 1 in map, → not found

Push mp[3] = 1

TC - $O(N)$

SC - $O(N)$

i = 2, x = 1

Find 4 - 1 = 3 in map → found

Optimal: (using 2 pointer)

First sort the array and try to chose numbers in a greedy way.

```
left → index = 0
right → index = n - 1
while (left < right)
{
  sum = a[left] + a[right];
  if (sum < target)
    left ++;
  if (sum > target)
    right--;
}
```

T.C - O(N)
S.C - O(N)

If we have to return index of 2 sum elements, then we will have to make a copy of array before sorting the original array.

3-Sum:

arr[] = {-1, 0, 1, 2, -1, -4}

Find triplets that sum up to value = 0

arr[i] + arr[j] + arr[k] = 0

(i != j != k)

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

Brute Force - Using 3 loops

```
(i = 0 → n)
  (j = i+1 → n)
    (k = j+1 → n)
      if (arr[i] + arr[j] + arr[k] == 0)
      {

      }
```

T.C - O(n^3)
S.C - 2K (no. of triplets)

We will store the triplets in set, so that only unique triplets are stored. Before storing sort the triplet.

Better - (Using Hashing)

$arr(k) = -(arr[i] + arr[j])$

We will store only those elements in the map, which are between i and j pointers to shorten our search space.

```
for (i=0 to n)
 {
  set<int> hashset;
  for (j=i+1 to n)
   {
    int third = -(arr[i]+arr[j]);
    if (hashset.find(third) != hashset.end()) {
      vector<int> temp = {nums[i], nums[j], third};
      sort(temp.begin(), temp.end());
      ans.insert(temp);
    }
    hashset.insert(nums[j]);
  }
}
```

TC - $O(n^2) * O(\log M)$

SC - $O(N) + O(\text{No. of triplets}) \times 2$

Optimal:

Using 3 pointer approach

Wherever sum < 0, increment j

sum > 0, decrement k

Wherever j > k, stop and increment i to next unique element in the array.

Sort the array beforehand.
TC - $O(n \log n) + O(N^2)$
SC - O(no. of triplets)

Initially $j = i+1$ & $k = n-1$.

Whenever we find a triplet, increment and decrement j & k until a unique element is found

1 → arr [-2 -2 -2 -1 -1 -1 0 00 2 2 2 2]
i j k

2 → arr [-2 -2 -2 -1 -1 -1 0 00 2 2 2 2]
i j k

.....

4 Sum:

$nums[i] + nums[j] + nums[k] + nums[l] = target$

$[i != j != k != l]$

arr[] = { 1 0 -1 0 -2 2 } target = 0

[-2 -1 1 2], [-2, 0, 0, 2], [-1, 0, 0, 1]

Brute Force: Using 4 loops

Calculate sum with proper typecasting otherwise it will overflow limit of integer.

```
long long sum = a[i] + a[j];
sum += a[k];
sum += a[l];
Then compare with target
```

TC - O(n^4) SC - O(no. of quads) x 2

Better: (Hashing)

$nums_2[l] = target - (nums_2[i] + nums_2[j] + nums_2[k])$

We will store elements b/w j & k in the map

TC - O(n^3) * O(log m)

SC - 2x O(no. of quads)

Optimal: Sort the array target = 8

arr[] = [1 11 222 333 4 44 5 5]

Same working as 3 Sum rule but this time we will use 2 outer loops i.e. i and j will remain fixed and k and l will iterate towards each other. Terminating condition will be till (k<l) (k>=l). Whenever k>=l, j will move to next unique element. Then when j reaches end of array, i will move to next unique element. j will reset to i+1 index.

k -> j+1 l -> n-1

k++ l--

TC - O(n^2)

SC - O(no. of quads)

Longest Consecutive Subsequence ⇒ *

Find length of longest sequence which contains the consecutive elements.

arr[] = {100, 200, 1, 3, 2, 4}

O/P → 4

Exp → 1, 2, 3, 4 is longest

arr[] = {3, 8, 5, 7, 6}

O/P → 4

Exp → 5, 6, 7, 8 is longest

Brute: Use two loops and find next element of each element, if it exists then increase length, and find its next element further.

arr[] = {102, 4, 100, 1, 101, 3, 2, 1, 1}

Choose a no. and find its next element

```
for(i=0 to n)
{
  x = arr[i];
  cnt = 1;
  while(search(arr, x+1) == true)
    {
      x = x+1;
      cnt++;
    }
}
```

TC - O(N²)

SC - O(1)

Better: Sort the array first, longest = 1. (stores length of longest seq.)

cntCur = 0 last_smaller = INT_MIN (stores last element and checks whether curr element is sequence of last or not)

We start fresh if last_smaller is not equal or one less than curr.

[1, 1, 1, 2, 2, 2, 3, 3, 4, 100, 100, 101, 101, 102]

longest = 4

cntCur = 0

1 2 3 4

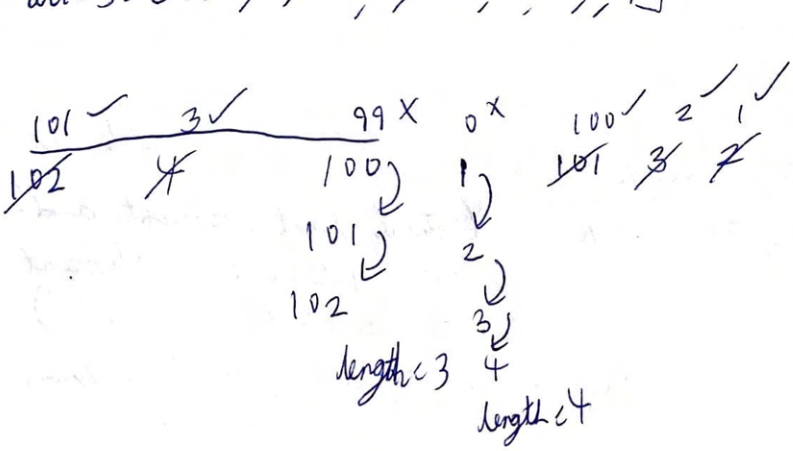
```
for(i=0 → n)
{
    if (arr[i] - 1 == lastSmaller)
    {
        cnt++;
        lastSmaller = arr[i];
    }
    else if (arr[i] == lastSmaller
    {
        continue;
    }
    else {
        cnt = 1;
        last_Smaller = arr[i];
    }
    longest = max(longest, cnt);
}
```

TC - O(N log N)
SC - O(1)

Optimal -

i) Put everything in set data structure (unordered_set)
ii) Start iterating elements in set
- For each element, we search for its (-1) element, if it exists then the curr element is not starting point of sequence
- If it doesn't exist, then it may be a starting point, then we keep on searching its +1 elements until found.

arr[] = [102, 4, 100, 1, 101, 3, 2, 1]



TC - O(N) in worst case for set
O(1) in average case + O(2N) for all elements

SC - O(N)

Longest Subarray with Sum K

1) N = 8 arr[] = {6, -2, 2, -8, 1, 7, 4, -10} K = 0

O/P → 8

Explanation → Subarrays with sum 0: {-2, 2}, {-8, 1, 7},
{-2, 2, -8, 1, 7}, {6, -2, 2, -8, 1, 7, 4, -10}

Length of longest subarray = 8

2) N = 3 arr[] = {1, 0, -5} k = 0

O/P → 1

Subarray: {0} length = 1

Brute Force -

Generate all subarrays and calculate sum of each subarray.

```
for(i = 0 → n)
for(j = i → n)
{ sum = 0;
  for(k = i → j)
    s += a[k]
if(s == target)
  len = max(len, j - i + 1)
}
}
```

TC - $O(n^3)$ SC - $O(1)$

Optimised Brute -

```
for(i = 0 → n)
{ sum = 0
for(j = i → n)
{
  s += a[j];
}
if(s == k) len = max(len, j - i + 1)
}
}
```

TC - $O(n^2)$.
SC - $O(1)$

Better - (Using hashing)

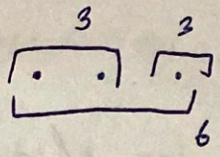
arr[] = { 1, 2, 3, 1, 1, 1, 1, 4, 2, 3 } k = 3

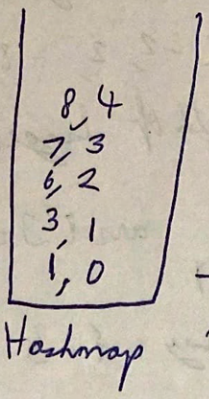
Initially at first ele, sum = 0

i = 0, sum = 1 len = 0

i = 1, sum = 3 len = 2.
here (sum == k)

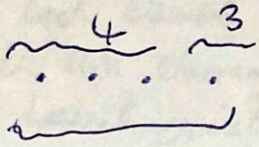
i = 2, sum = 6





Till index 1, we got 3, and again at index 2 we got 3, which means another subarray with sum == k exist in the array, but as its len = 1, we will not change the length.

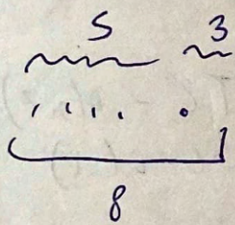
i = 3, sum = 7



For the subarray sum = 3, remeare previous sum will have to be 7-3 = 4, we will check whether any sum = 4 exists in map or not. If not, put it in hashmap,

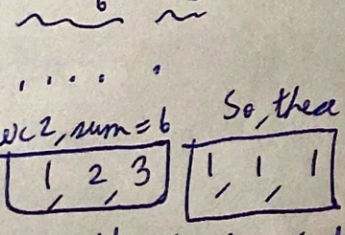
If yes, update length if greater.

i = 4, sum = 8



we will check whether 8-3 = 5 exist in map. It does not exist

i = 5, sum = 9



9-3 = 6 exists in hashmap, and index = 2

So, thea remaining == target

Currently i = 5, index where sum = 6 is 2, so 5-2 = 3 is the new length of the subarray.

```
mp<long long, int> mp;
long long sum = 0;
int maxLen = 0;
for(int i = 0; i < n; i++){
    sum += a[i];
    if(sum == k)
        maxLen = max(maxLen, i+1);
    long long rem = sum - k;
    if(preSumMap.find(rem) != mp.end()){
        int len = i - mp[rem];
        maxLen = max(maxLen, len);
    }
    if(mp.find(sum) == mp.end())
        mp[sum] = i;
}
return maxLen;
```

TC - O(N log N) - ordered map
SC - O(N)

Optimal - 2 pointer

arr[] = { 1, 2, 3, 1, 1, 1, 3, 3 } k = 6

Initially sum = 1
Increment j
j = 2, sum = 3
sum < k
Increment j
j = 3, sum = 6
(sum == k)
len = j - i + 1; // 3 in our case

Now, again increment j
j = 4 sum = 7
sum > k
Increment i
& decrement a[i] from sum
again sum = 6
& len = 3

After calculating maxLen = 4

TC - O(2N)
SC - O(1)

```
int i = 0, j = 0;
long long sum = a[0];
int maxLen = 0;
while(j < n)
{
    while(i <= j && sum > k)
    {
        sum -= a[i];
        i++;
    }
    if(sum == k){
        maxLen = max(maxLen, j - i + 1);
    }
    j++;
    if(j < n) sum += a[j];
}
return maxLen;
```

Number of Subarrays with XOR k

arr[] = [4 2 2 6 4]

k = 6

100
^010
110 → 6

{4, 2}
{6}
{2, 2, 6}
{4, 2, 2, 6, 4}

xr → 4

~~4~~ 4, 2

Brute Force -

Generate all subarrays and calculate XOR

100
110
010 → 2

```
for (i = 0 → n)
{
int XOR = 0
for (int j = i; j < n; j++)
  { XOR = XOR ^ a[j];
  if (XOR == k) cnt++;
}
}
```

TC - $O(n^2)$ SC - $O(1)$

Optimal -

Using hashing) (Same as previous ques logic)

$x = XR \wedge k$

where XR is XOR of whole subarray

TC - O(N)

(4 2 2 6 4)

~~4~~

xr = 0

~~mpp[xr]++~~

xr = xr ^ a[i];

= 0 ^ 4 = 4

int x = 4 ^ 6 = 2

mp[x] ++;

(0, 1)

```
int xr = 0;
mpp[xr]++;
for (i = 0 → n) {
    xr = xr ^ a[i];
int x = xr ^ k;
cnt += mpp[x]; return cnt;
}
```

Longest Substring Without Repeating Characters →

s = "abcabcbb"

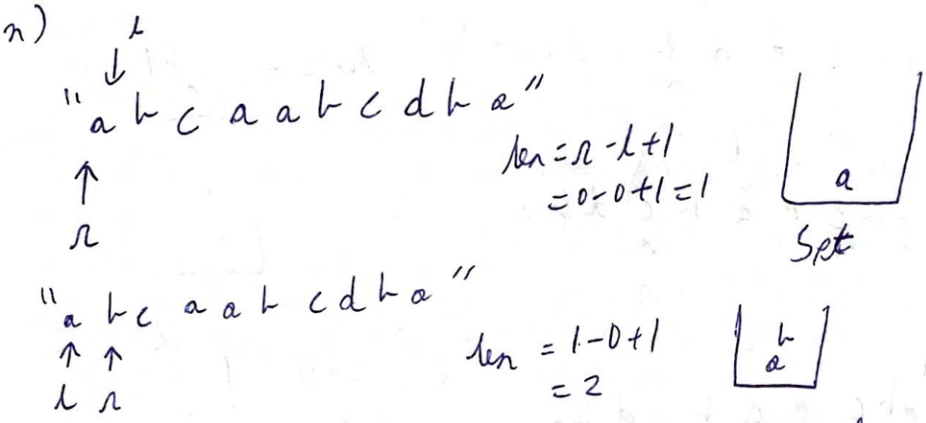
Ans - 3

Brute Force - Generate all substrings, and use a hashset for checking count of each character in a substring.

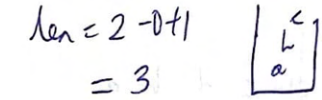
TC - $O(n^2)$

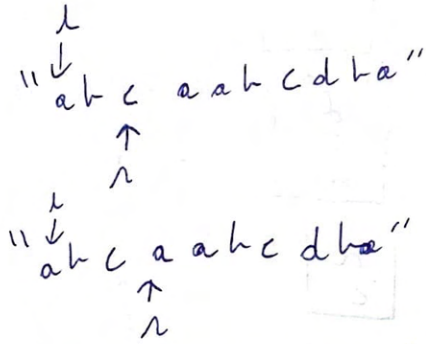
SC - $O(n)$

~~Optimized~~:



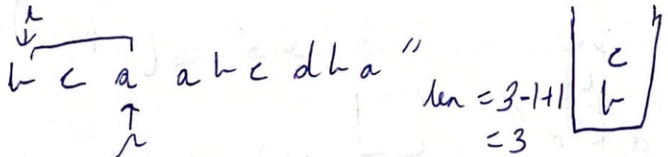
We will check whether char 'c' is present in set or not. If not, push it in set.





Here, 'a' is already in set. So, we will remove 'a' from set and increment l

(whichever element is first in set will be removed)



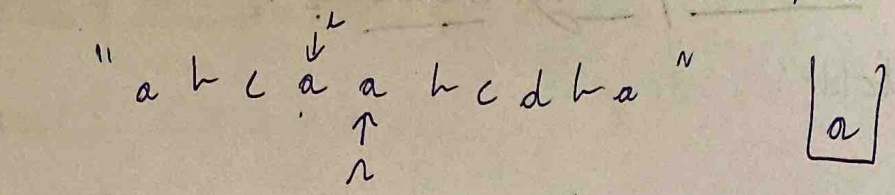
Now we check again whether present in set or not.

Not present, so push 'a'

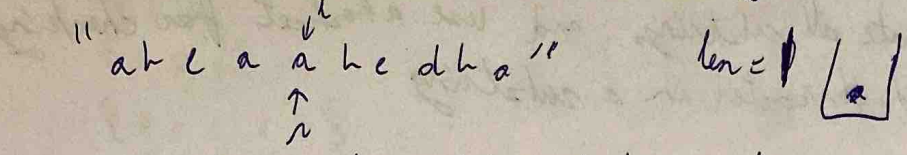
Again a is present in set,

so, we remove 'b' from set and then check again,

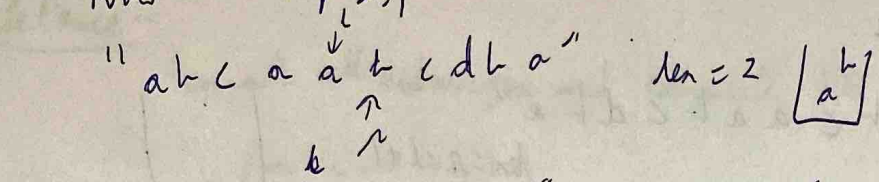
It still exists in set remove 'c' and increment l

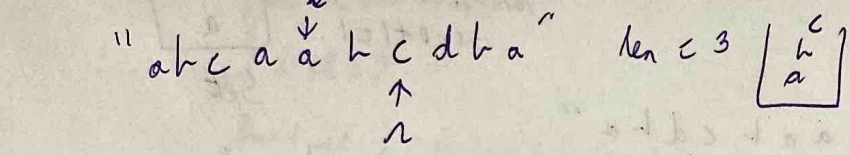


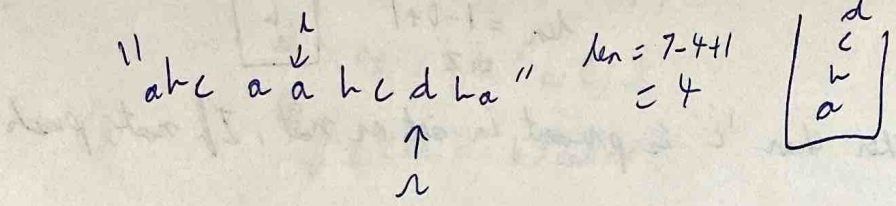
It still exist, remove a and increment l

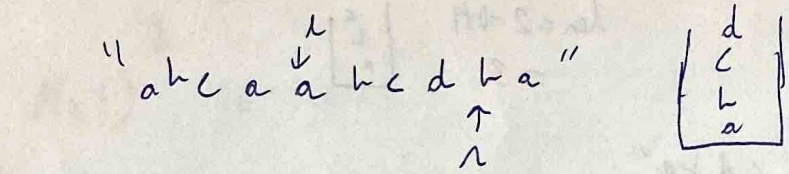


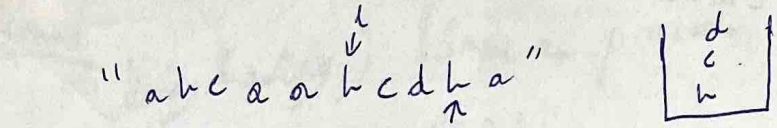
Now set is empty, push 'a' and increment r

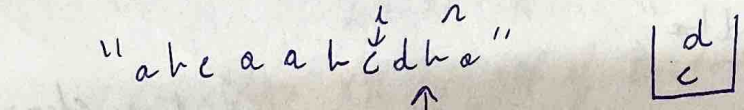


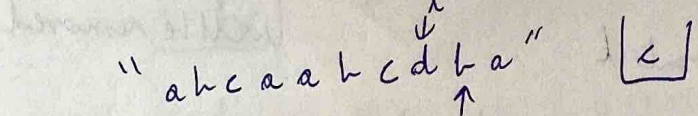


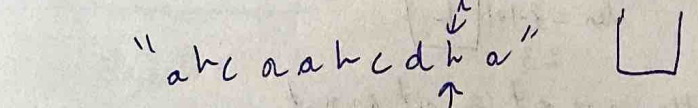


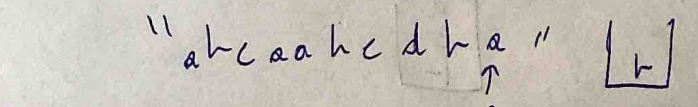


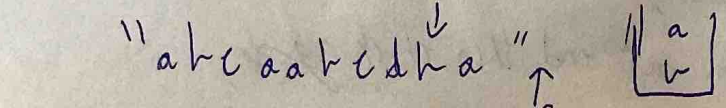












$T(-O(2N)$

$S(-O(N)$

Max length = 4

Optimised approach →

In previous approach,

```
   l   r
"abcabcdba"
```

I had to move 3 places again to traverse, and get new unique subarray. This is increasing our T.C.

So, to reduce this

Eg - abcdcefg

```
     l  r
```

To remove this repeating char from set, we had to move 5 places i.e. abcdc, which wasted a lot of time

We can reduce this, by storing the index along with the character,

E.g. we will use map

```
"abcabcdba"
 ↑   ↑
 l   r
```

(c, 2)
(b, 1)
(a, 0)

As we can see, a was found at index 0, if we move our l to index 0 + 1, then char will not repeat

```
"abcabcdba"
  ↑ ↑
  l r
```

(c, 2)
(b, 1)
(a, 3)

```
"abcabcdba"
  ↑   ↑
  l   r
```

Again, 'a' we will move l to 3+1 index

```
"abcaabcdba"
    ↓
    l
     ↑
     r
```

(c, 2)
(b, 1)
(a, 4)

In this way, we will solve further.

T.C - O(N)

S.C - O(N)

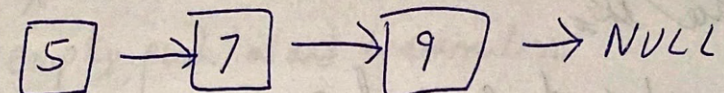
Merge two Sorted Lists →

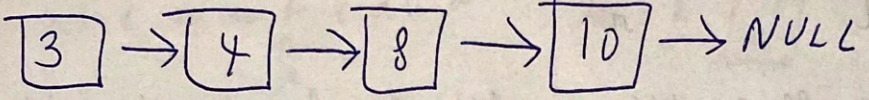
Brute Force - Using external space and merging it like you merge arrays

SC - O(N) TC - O(N)

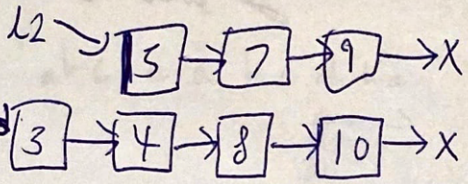
Optimal - Not using extra space (In-place)

TC - O(N)
SC - O(1)



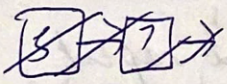


- Take 2 variables l1 & l2, ~~point~~ compare first element of both lists, whichever is smaller, point l1 to it

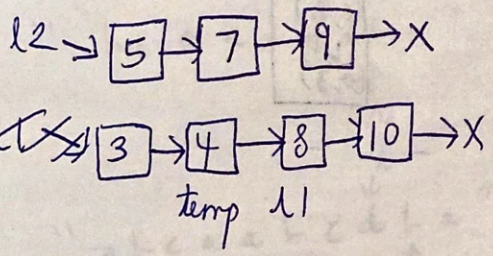


temp = null

tl = l1 → next until l1 < l2



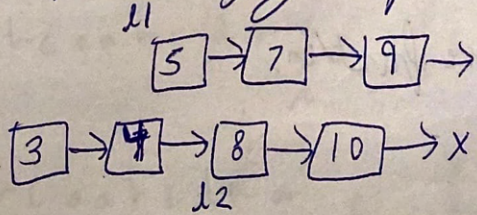
Before moving l1, assign temp = l1



Here l1 > l2,

We do temp → next = l2 , this breaks the chain and arranges the list.

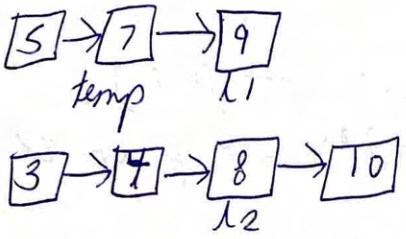
Now, after modifying temp → next, we swap l1 & l2.



Now, second iteration starts, temp = NULL
l1 = l1 → next until l1 < l2

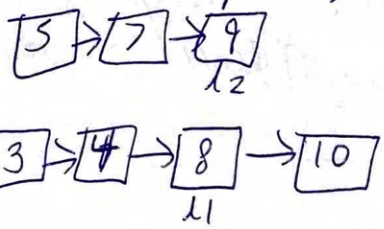
~~temp = l1~~

but before moving l1 to next, assign it to temp.

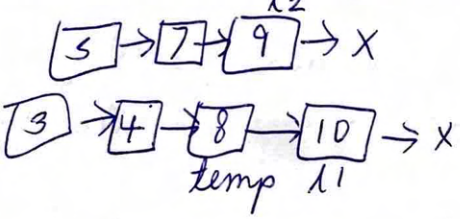


Now, again temp → next = l2

swap l1, l2

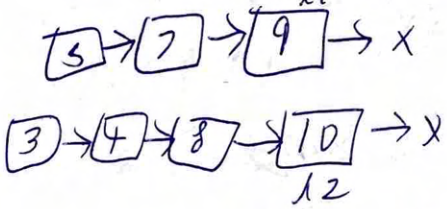


Third iteration begins

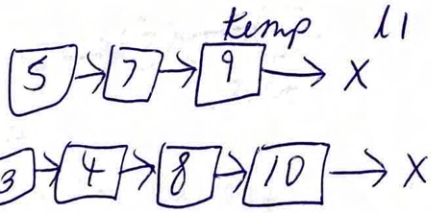


Now, l1 > l2, temp → next = l2

swap (l1, l2)



4th

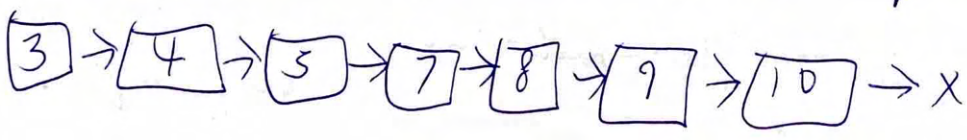


l1 == NULL //Its over here, temp →

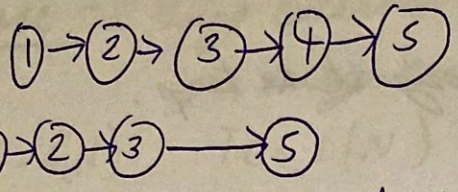
temp → next = l2;

~~swap (l1, l2)~~

l1 == NULL in 4th iteration, so loop breaks,



Remove N^{th} Node from End of linked list →



Naive - Calculate length of linked list, — O(N)
Subtract length - N ⇒, it will be 3rd node from the front in above case,

Now, using another loop cnt = 0, increment counter, when cnt = 3, we assign temp → next = temp → next → next

node = t → next (before assigning this)

delete (node)

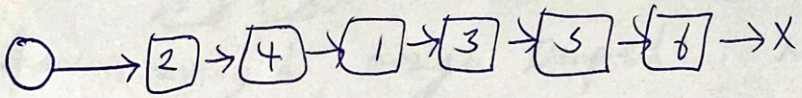
edge case, if N == length of LL,

node = head
head = head → next

TC - O(N) + O(N) — O(2N)

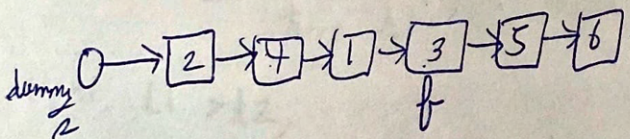
SC - O(1)

Optimal - Create a dummy node pointing to head

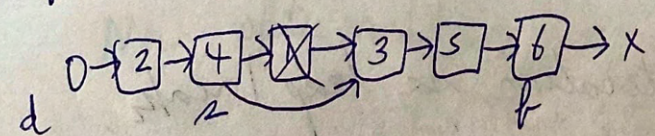


Initialize two pointers, fast, slow = dummy.

If N = 4, we move fast by 4 steps →



Now, move both fast & slow by one node (step) until fast → next == NULL (reaches last node)

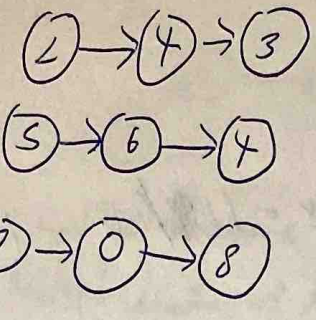


Assign slow → next = slow → next → next;

Return dummy → next;

TC - O(N)
SC - O(1)

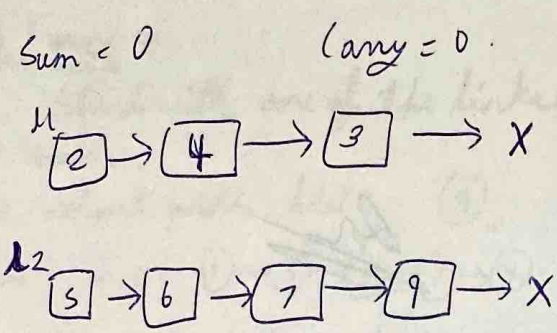
Add Two Numbers Given as Linked Lists



Digits are stored in reverse order.

Approach:

1) Create a dummy node, assign 0, assign a temp node to it. Dummy node will give head of the Linked list.



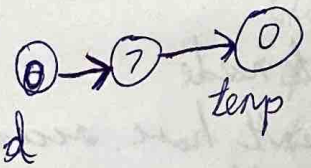
If l1 != NULL, sum += carry + l1

l2 != NULL, sum += ~~carry~~ l2

Move, l1, l2 & temp one step ahead

Again check l1 & l2 != NULL & add to sum

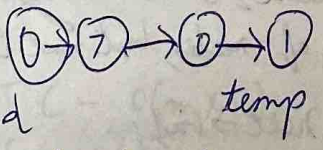
~~sum = 0~~ Here, 6+4 = 10, we store 10%10 in new node and store 10/10 in carry



carry = 1
sum = 0

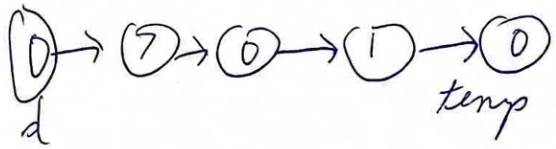
Move l1, l2 & temp one step

l1 = 3 l2 = 7 sum = 0 3+7+1 (carry from prev)
= 11
sum %10 = 1
sum/10 = 1

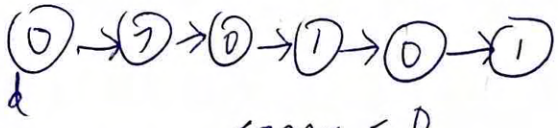


Move l1, l2 & temp

At this moment l1 == NULL
But l2 is not null
add l2 to sum
sum += l2 + carry
= 9 + 1
= 10 carry = 10/10 = 1



Now move l2 & temp
Now, l2 == NULL
But carry != NULL
So, sum += carry = 1



carry = 0
Increment temp to next,
Now all l1, l2 & carry == NULL
Return dummy->next,

SC - O(N)
TC - O(Max(l1, l2))

Delete Given node in a Linked List

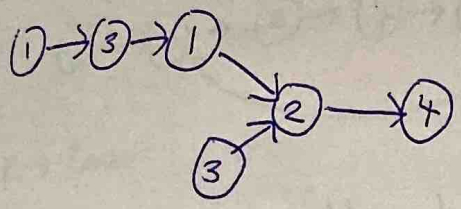
Delete node
We don't have access to head of LL.

It is given that node will never be a tail of the LL.

Approach: We can only access nodes in the right of node, not on left, we will modify value of nodes, but not the nodes.
So, we do node->val = node->next->val;
node->next = node->next->next;

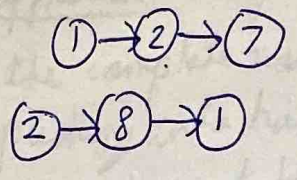
TC - O(1) SC - O(1)

Find Intersection of Two Linked Lists →

E.g- 

O/P → 2

Note: It is not the same value that we have to find in the linked lists, we have to find same node.

E.g-2 

O/P → NULL

Brute Force →

Start with one of the linked lists and check for all of the nodes,

- We start with list2 ③, check it with all nodes in list1, i.e. ①→③→①→②→④ No node is same.
- Move to next node in list 2
- Now in list 2 ②, check with all nodes in list1 i.e., ①→③→①→②→④

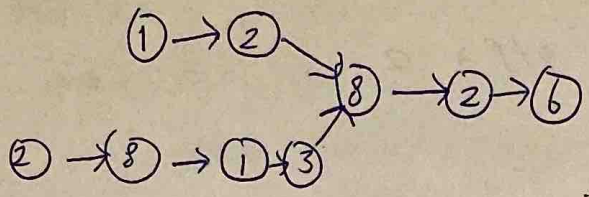
Here, we find node ② is same. We return ②.

- If we exceeded second linked list, and did not find any intersection return NULL.

```
Code →  while (head2 != NULL)
        {
          node* t = head1;
          while (t != NULL) {
            if (t == head2) return head2;
            t = t → next;
          }
          head2 = head2 → next;
        }
        return NULL;
}
```

TC - O(m×n)
SC - O(1)

Better Approach - (Using Hashing)



1) We traverse list1 and hash the node address.
2) Now iterate list2 and check for each node, whether node is present or not in hash table

TC - O(N) + O(M)
SC - O(N)

1st Optimal approach →

1) Using 2 dummy pointers, calculate length of both lists
In above case, list 1 = 5, list 2 = 7
2) Find difference b/w 2 lists length.
diff = 2 (larger - shorter)
3) Again start from head of both linked lists.
4) Move dummy2 2 spaces ahead. (of larger linked list)
5) Then start traversing both linked list simultaneously.
6) The place/node where they collide will be the intersection.

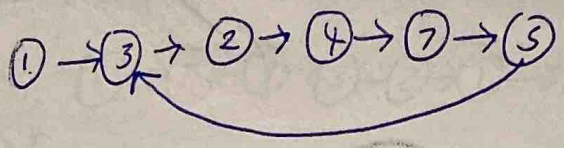
TC - O(M) + O(M-N) + O(N)
= O(2M)
SC - O(1)

SC - O(1) TC - O(2M)

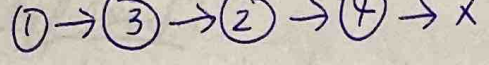
2nd Optimal approach →

(Short code) 1) Using d1 & d2 start with both lists simultaneously until either of them reaches end of list.
2) If suppose d1 reaches end first, assign it to list2's head, otherwise assign d2 to list1's head if it reaches first.
3) Then start traversing again, now the second dummy ptr will reach end, assign it to list1 or list2 accordingly.
4) Start traversing again. The place/node where they collide will be the answer.

Detect a Cycle in Linked List



o/p → True



Naive approach: (Using Hashing)

Hash the complete node, while traversing the list, simultaneously before pushing into hash table, check whether node is already present or not in table.

SC - O(N) TC - O(N)

Optimal -

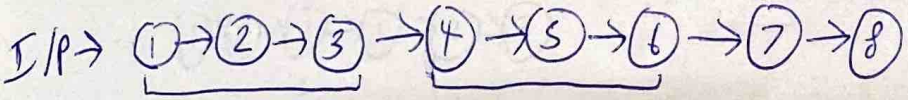
Using fast and slow pointers, fastptr moves by 2 nodes, while slow ptr moves by 1 node.

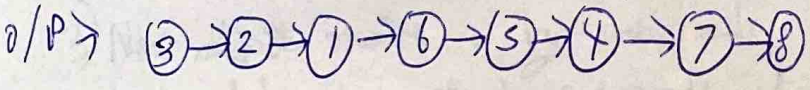
If slow & fast ptr meet, then there is a cycle in the list.

TC - O(N) SC - O(1)

Reverse a Linkedlist in groups of size k *

k = 3





If k = size of remaining elements < k, don't reverse

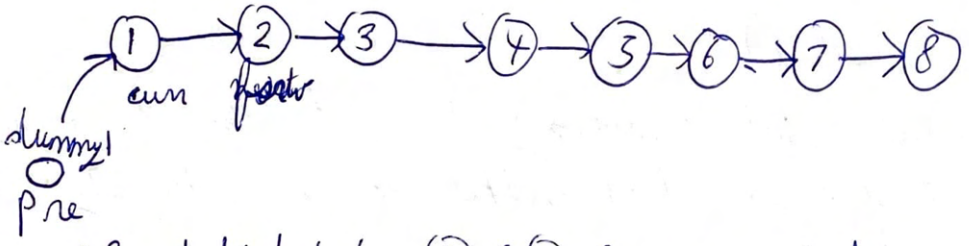
Approach:

1) Count the length of linked list
   Here cnt = 8 & k = 3
2) Now, we know we'll be having 8/3 i.e. 2 groups of size 3
3) We need to perform min 2 operations within a group of size 3

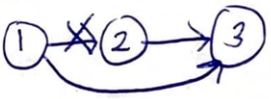
One to reverse this link Another to reverse this link

✓

4) Create a dummy node and point it to head of LL. Assign a pre ptr to it.



a) Break link b/w (1) & (2) & make a link b/w (2) & (3)

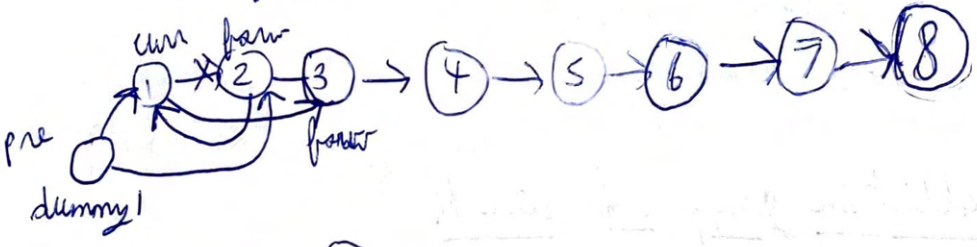


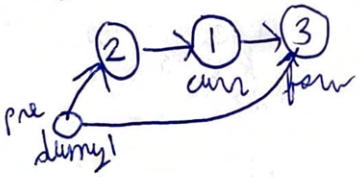
curr → next = fowr → next

fowr → next = pre → next

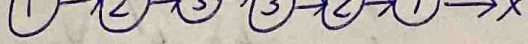
pre → next = fowr;

fowr = curr → next





Check if Palindrome Linked List →



O/P → True

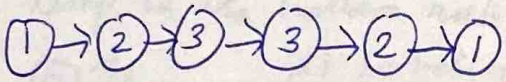
Naive approach:

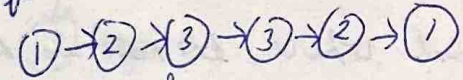
Store all elements in a vector, and then check for palindrome.

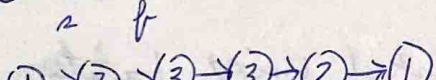
TC - O(2N)
SC - O(N)

Optimal approach:

1) Find middle of list
2) Reverse the right half of the list

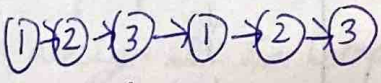






Skip when fast → next → next == NULL

Now reverse right half



3) Now ~~start~~ move s by one node, then start traversing both s & dummy ptr by one node, if they match and s reaches end of LL, Then it is a palindrome.

TC - O(N/2 + N/2 + N/2)
SC - O(1)

Find starting node of cycle in Linked List →

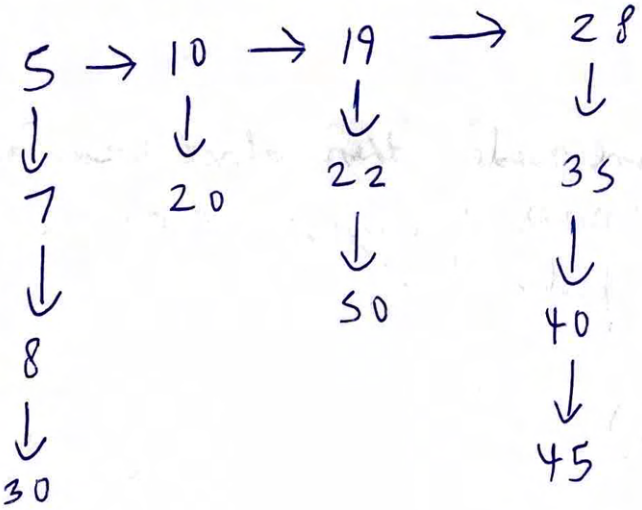
Perform same steps used to find a loop.
When a loop is detected, move slow ptr to head again
Then move slow & fast by one node at a time.
The place where they meet/collide again will be the starting point of LC.

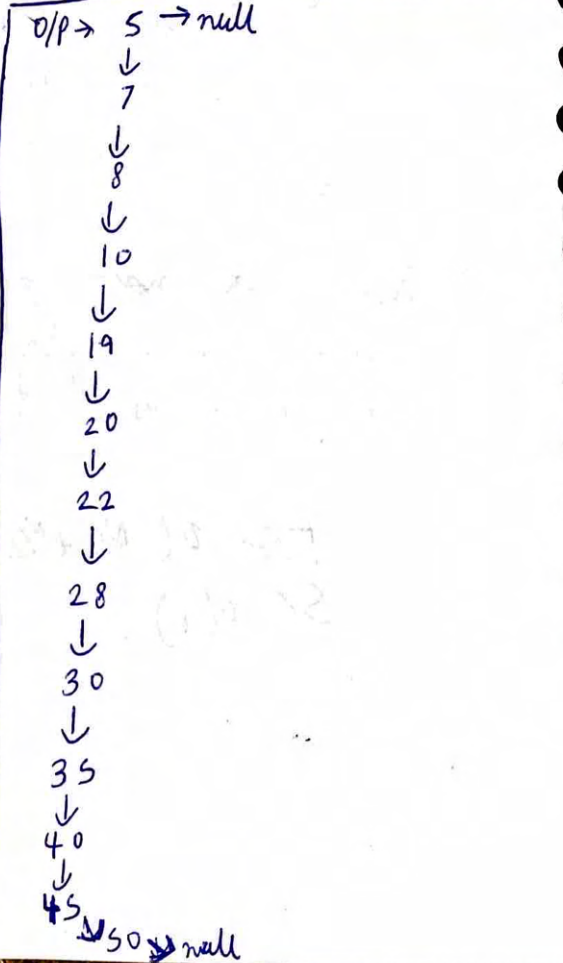
TC - O(2N)
SC - O(1)

Flattening of a Linked List →

- Linked List of size N
- Every node represents a linked list and contains 2 ptrs of its type:
  - next ptr to the next node
  - bottom ptr to a linked list where the node is head

Note: Flattened list will be printed using bottom ptr instead of next ptr.

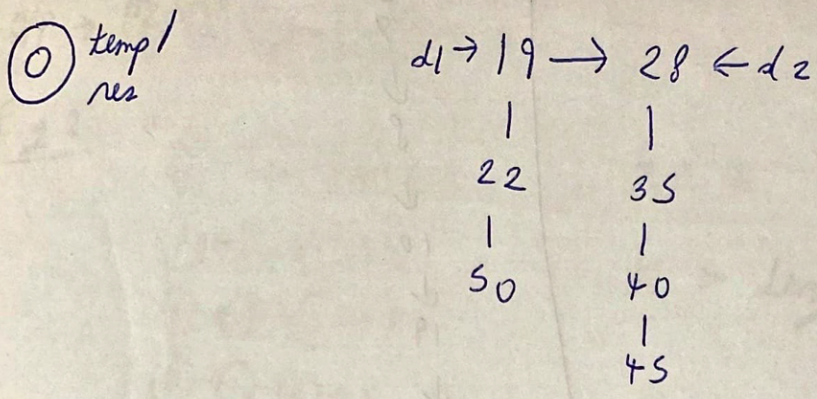




Approach:

Merging two, or more linked list into one is the task ~~we'll~~ we'll be going to perform here.

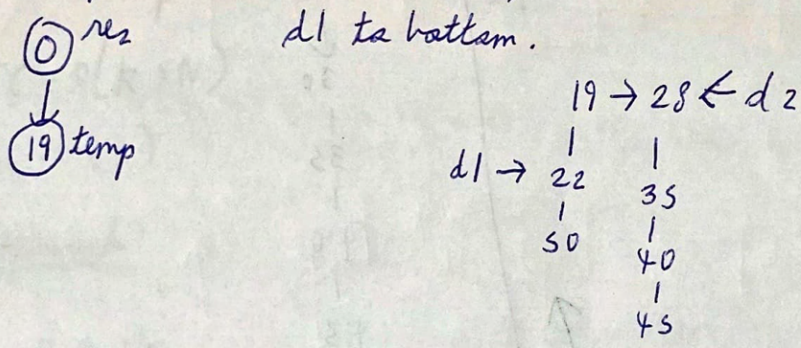
1) Create a dummy node, assign temp & res to it



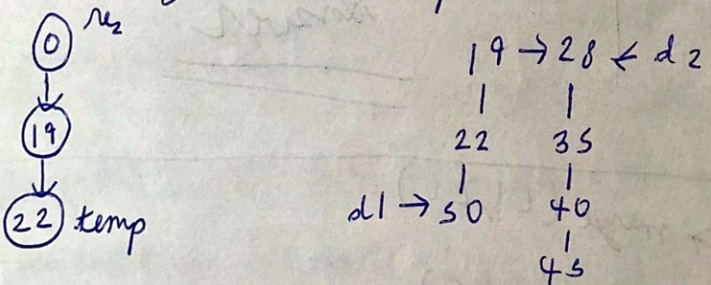
We keep 2 pointers one at 19 & other at 28

Whichever is smaller, add it to temp's bottom.

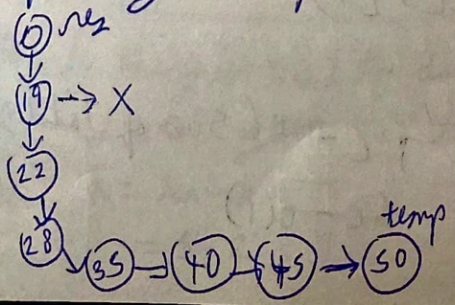
Move temp to its bottom node, since 19 has been taken move d1 to bottom.

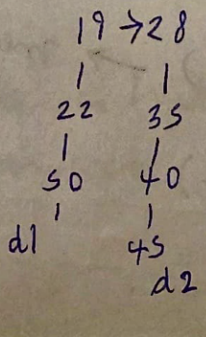


Now again perform same step, compare d1 & d2, whichever is smaller assign it to temp's bottom, and move d1 & temp



After performing all steps

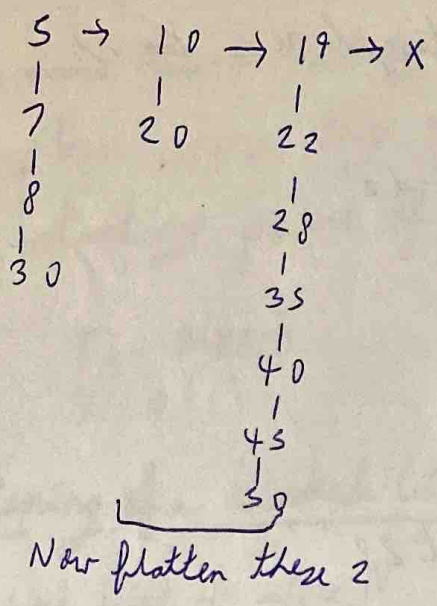




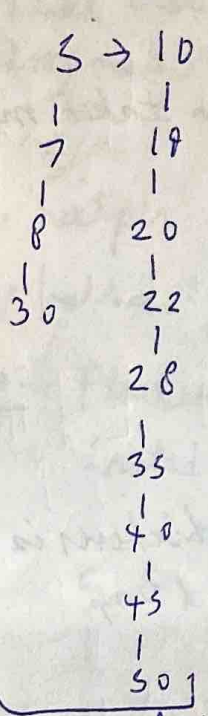
Both d1 & d2 == NULL

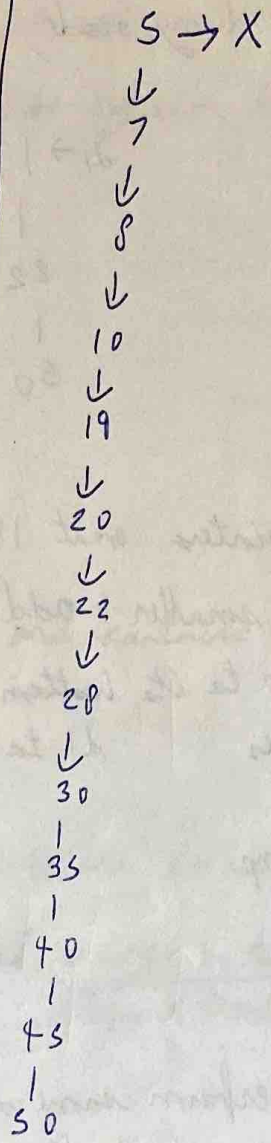
Bottom ptr of res is head of the answer linked list. Assign its next to NULL

We have flattened last 2 LL into 1



Now flatten these 2





Answer

L6 flatten(L1) → merge(L1, L6)
flatten(L2) → merge(L2, L5)
L5 flatten(L3) → merge(L3, L4)
L4 flatten(L4)

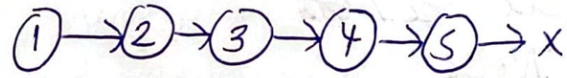
T.C. - O(Sum of total no. of nodes)

S.C. - O(1)

Rotate a Linked List →

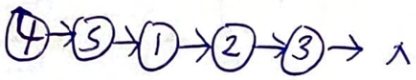
Given head of list, rotate list by k places (right rotate)

E.g 1

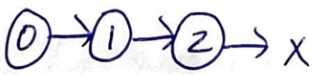


$K = 2$

O/P →

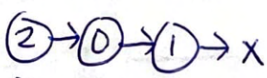


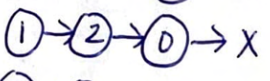
E.g 2

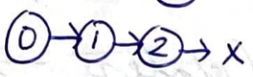


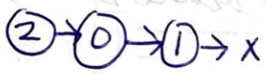
$k = 4$

$k >$ length of linked list









Naive Approach :

Pick up the last node, put it at the front

TC - $O(k \times N)$

SC - $O(1)$

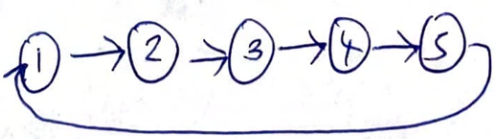
Optimal approach :

$k < len$ $k >= len$

If len = 5, k = 12

If $k >= len$ then $k \% len$

1) Find length of the Linked List, we will use a ptr to calculate length. When ptr reaches last node, connect it to the head of the list



The only thing we need to do now is break the link b/w ③ & ④ it will give us the answer.

So, to do that $k = len - k$

$= 5 - 2 = 3$

Now again take a ptr, point it to head, start traversing until you don't reach the k^{th} node,

head = 3'rd node → next

3'rd's → next = NULL;

This is the final list

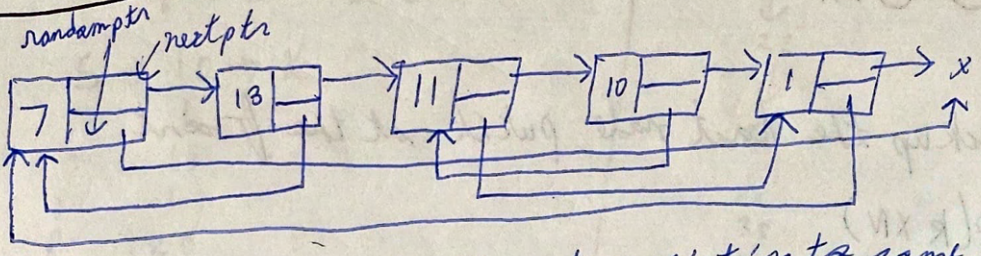
④ → ⑤ → ① → ② → ③

TC- $O(N) + O(N - N\%K)$

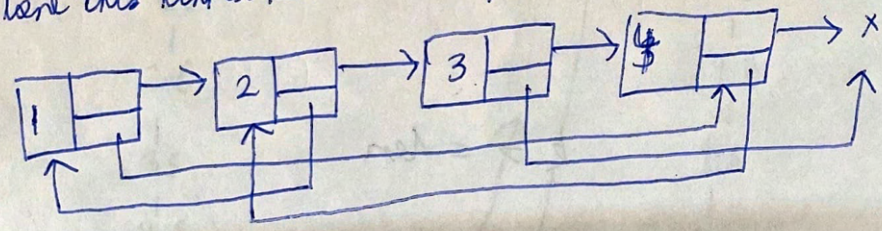
$= O(N)$

SC- $O(1)$

Clone Linked List with Random and Next Pointer →

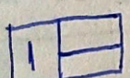


Clone this linked list with all pointers pointing to same nodes



Brute Force-

- Create a hashmap <node, node>
- Start traversing the linked list, make a same node



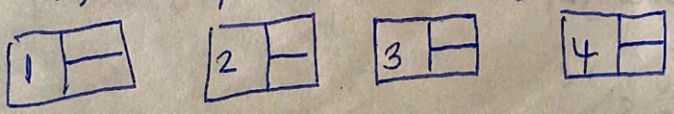
, but right now don't assign any pointers

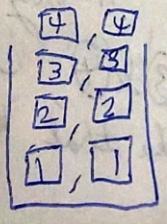
put both original node & new node in map.

<Node, Node>

- Increment the ptr.
- Create a copy of next node & push in map

Similarly, we keep doing this and get

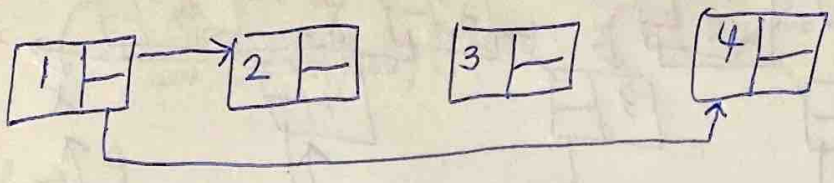




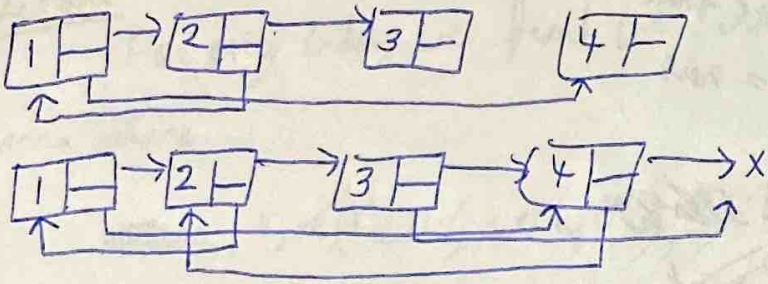
Now traverse again the original list, we go on 1 and see what is its next pointer pointing to.

Point next ptr to 2

and point random ptr to 4



Now go to next node in list



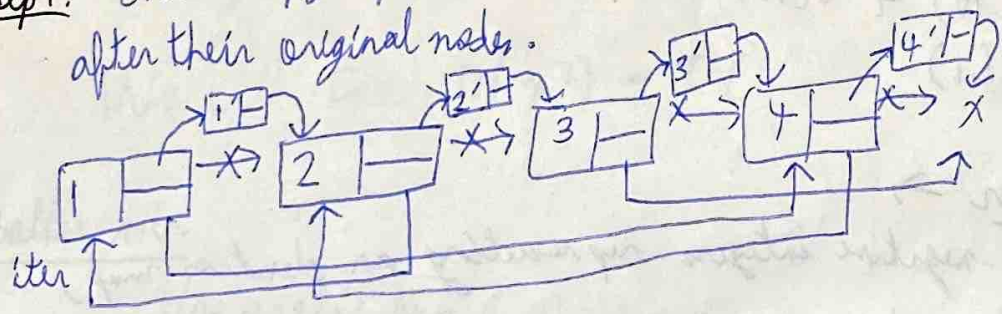
Return head;

TC - O(N) + O(N)

SC - O(N)

Optimal approach:

Step 1: Create copy of nodes and place them ~~at original position~~ right after their original nodes.



Step 2: Point the random ptrs "iter" (ptr pointing to the start with a variable ~~iter~~ head of list)

As we can see, iter's → random → 4

Then iter's → next → random = iter's → random → next

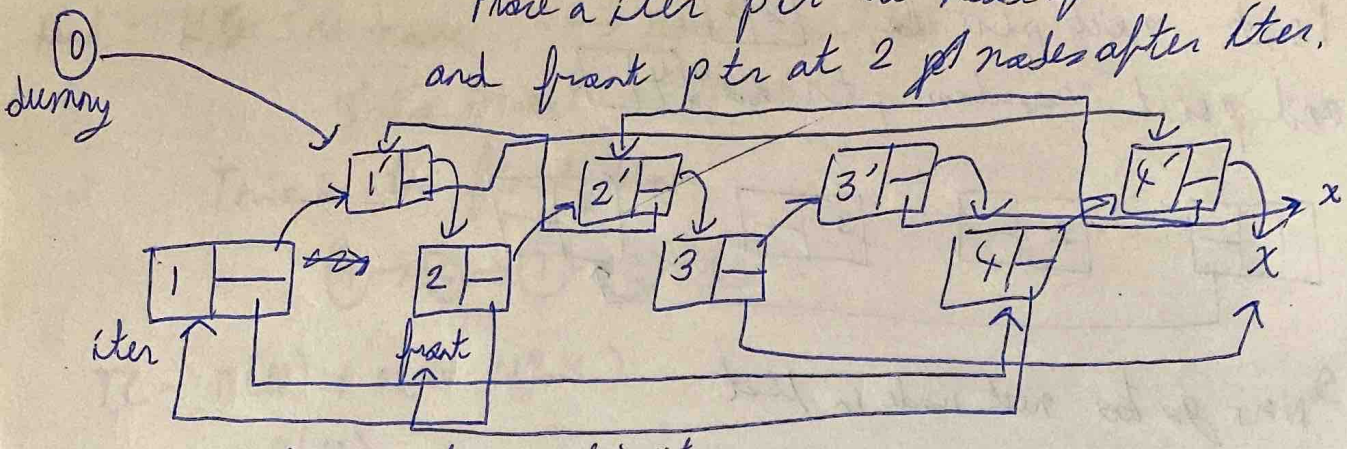
Move iter by 2 steps

Again perform above step until (iter != NULL)

Step 3: Separate the original list and deep copy list

Take a dummy variable, value 0.

Place a iter ptr at head of list and front ptr at 2 nodes after iter.



front = iter → next → next

dummy's next = iter's next

iter's next = front

~~iter = iter → next~~

~~front = front → next → next~~

dummy = dummy → next

iter = front;

~~Do~~ Do this in loop

TC - O(N) + O(N) + O(N)

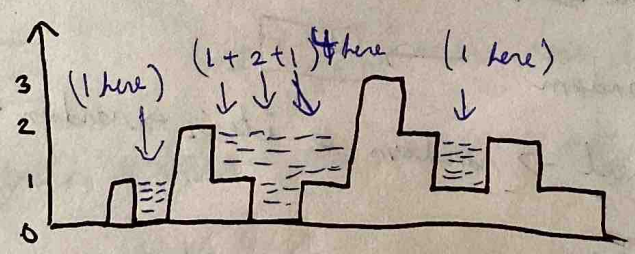
SC - O(1)

Trapping Rainwater →

n non-negative integers representing an elevation map, width of each bar = 1

Compute how much water it can trap after raining.

E.g-1

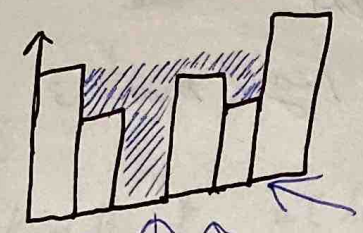


So total 6 units

I/P → height: [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]

O/P → 6

Ex-1.2 IP → [4, 2, 0, 3, 2, 5]



(2here) (4here) (1here) (2here) = 9 units

O/P → 9

Brute Force:

For every index, we find we find unit of water that's gonna store.

~~Water~~ ~~Max(building at left, building at right) - current height~~

Min(left[i], right[i]) - a[i]

where left[i] is the max in left of current element
and right[i] is the max in the right of other, current element

This will take $O(n^2)$ - TC

Better soln

Use prefix sum

Make a prefix max & suffix max array

pre → 0 1 1 2 2 2 2 3 3 3 3 3

suffix → 3 3 3 3 3 3 3 3 2 2 2 1

pre is keeping max towards the left

suffix is keeping max towards right

With this we can remove the nested loop, where we were finding max ele in left & max ele in right.

E.g. for 3rd index, left max = 2 & right max = 3

T.C - $O(N)+O(N)+O(N)$ — 3 loops

S.C - $O(N)+O(N)$ — for 2 extra arrays

Optimal Soln →

Approach → left = 0, right = n-1, trapwater = 0

leftMax = 0 & rightMax = 0

```
if (a[left] <= a[right])
{
// If yes then compare a[left] with leftMax,
   whether a[left] > leftMax or not
```

[0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]

left — right

Here a[left] ≯ leftMax

leftMax remains 0, and increment left ptr,

Again do this comparison,

```
if (a[left] <= a[right])
{
   if (a[left] > leftMax)
   leftMax = 1

   left++;
}
```

At index 3, a[left] <= a[right] is true

So, we check

```
if (a[left] < leftMax)
   here it is not,
```

So, we run the else case

```
else {
    trapWater += (leftMax - a[l]);
}
}
```

Now at index 4, a[left] > a[right]

perform same steps as above

```
if (a[right] >= rightMax)
    rightMax = a[right];
else
    res += (rightMax - a[r]);
right--;
}
```

Intuition → In brute force, formula was $\min(leftMax, rightMax) - a[i]$.

Over here, either we take leftMax - a[i] or RightMax - a[i]

TC - O(N) SC - O(1)

Container with Most Water →

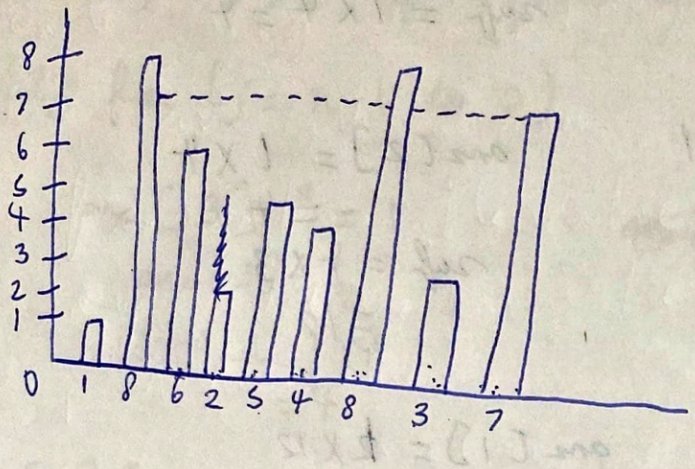
Given → array height of length n

n vertical lines drawn such that two endpoints of the i^{th} line are $(i, 0)$ and $(i, height(i))$;

Find 2 lines that together with x-axis form a container, such that the container contains the most water.

Return max amt of water a container can store.

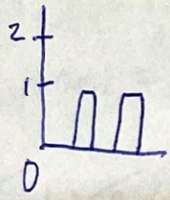
Eg -



I/P → [1, 8, 6, 2, 5, 4, 8, 3, 7]

O/P → 49

Eg-2



I/P → [1, 1]

O/P → 1

Brute Force -

From each index, calculate area with every other index, just like generating subarrays

Using two for loops

TC - O(n²)

Optimal - Using 2 pointer approach, l & r

[6, 4, 2, 5, 4, 6, 1, 3, 5]

l ↗ ↖ r

~~for~~ breadth = r - l;

minheight = min(a[l], a[r]); // 8 // 5

area = breadth * minheight; //area = 40

maxArea = max(area, maxArea) //maxArea = 40

→ In in L & r, whichever has small height, we will move that pointer

TC - O(N) SC - O(1)

Max Consecutive 1's

[1, 1, 0, 1, 1, 1]

O/P → 3

Approach: We will start traversing the array from 0 to n-1. And we will maintain a cnt = 0 & ans = 0 variable. Whenever we keep getting 1 in the array, we will increment cnt variable. For e.g. at index 1, cnt will be 2, because 2 1's have occurred till that index.

Now, if a new value occurs in the array we update the ans variable by comparing it with its previous value.

ans = max(ans, cnt);

And reinitialize cnt = 0;

After getting out of the loop we will again check above condition

ans = max(ans, cnt);

TC - O(N)

SC - O(1)

Remove Duplicates from Sorted Array II →

Each element can appear at most twice

e.g. - nums = [1, 1, 1, 2, 2, 3]

O/P → 5, nums = [1, 1, 2, 2, 3, _]

Eg 2- nums = [0, 0, 1, 1, 1, 2, 3, 3]

O/P → 7 nums = [0, 0, 1, 1, 2, 3, 3, _]

Approach:

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

i = 0 & j = i + 1

```
if (a[i] == a[j] && first)
{
  i++;
  j++;
  first = false;
}
```

if (i+1 == j) continue;
else a[i+1] = j

~~else if (a[i] == a[j] && !first)~~
~~{~~

Now, [1, 1, 1, 2, 2, 3]

```
if (a[i] == a[j] && !first)
{
  j++;
}
```

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

```
if (a[i] != a[j])
{
  a[i+1] = a[j];
  i++;
  j++;
  first = true;
}
```

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

A[i] == A[j]

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

[0, 0, 1, 1, 1, 2, 3, 3]

[0, 0, 1, 1, 1, 2, 3, 3]

[0, 0, 1, 1, 1, 2, 3, 3]

[0, 0, 1, 1, 1, 2, 3, 3]

[0, 0, 1, 1, 1, 2, 3, 3]

[0, 0, 1, 1, 2, 3, 3]

[0, 0, 1, 1, 2, 3, 3]

[0, 0, 1, 2, 3, 3]

[0, 0, 1, 2, 3, 3, 2, 3, 3]

↑ o/p → 7

Return i+1

N meetings in One Room →

N meetings in the form of (S[i], F[i]), where S[i] is the start time of meeting i and F[i] is the finish time of meeting i,

There is only one room in a form.

What is the max no. of meetings that can be accomodated in the meeting room?

Note: Starting time of one chosen meeting can't be equal to the end time of the other chosen meeting.

E.g - N = 6

S[] = {1, 3, 0, 5, 8, 5}

F[] = {2, 4, 6, 7, 9, 9}
1 2 3 4 5 6

Give o/p in the order of 1 based indexing as

Output : 1 2 4 5

1 meeting's interval - [1-2]
2 meeting's interval - [3-4]
4 meeting's interval - [5-7]
5 meeting's interval - [8-9]

So, 4 meetings can be held in this time interval.

Brute Force / Greedy Approach

What if I perform as many meetings possible in one room, and that is only possible if we perform meetings whose finishing time is early than starting of other

Take a vector of 3 quantities, i.e. start time, finish time, and their position of meeting.

Then sort the vector based on least finishing time (If same finish time, the with the first index first)

S[] = 1 0 3 8 5 8

F[] = 2 6 4 9 7 9

1 2 3 4 5 6

Store it in vector,

[[1,2,1], [3,4,3], [0,6,2], [5,7,5], [8,9,4], [8,9,6]]

Initially room is empty, so first meeting can take place, simply

[1,2,1] ✓

print 1

(1) end time of this meeting = 2.

Now, go to next meeting, its starting >= ending of previous.

So, here we can also do meeting 3.

[3,4,3] ✓

(1) (3) Now, end time = 4

Next meeting starts at 0, so it is not possible [0,6,2] X

Again, end time will remain same, i.e. 4

Next meeting starts at 5, i.e. [5,7,5] ✓

We schedule it and end time becomes 7

(1) (3) (5)

Next meeting starts at 8, i.e. [8,9,4], so we can perform this meeting. Now, end time = 9

(1) (3) (5) (4)

Next meeting starts at 8, so not possible.

So, once we have traversed all the meetings in the sorted order of their finishing times, we will get max no. of meetings that can be performed in one room.

So,

TC → O(N) for traversing S[], N[] & putting it into data structure

+ O(NlogN), as we are gonna sort this according to their finishing times.

+ O(N) for traversing through the data structure.

So, TC → O(NlogN)

SC → O(N) for data structure.

Minimum number of platforms required for a railway →

Two arrays that represent the arrival and departure times of trains that stop at the platform. We need to find the min no. of platforms needed at the railway station so that no train has to wait.

Eg:- N=6,

arr[] = {9:00, 9:45, 9:55, 11:00, 15:00, 18:00}

dep[] = {9:20, 12:00, 11:30, 11:50, 19:00, 20:00}

Evaluation -

Time interval	Platform
9:00 - 9:20	1
9:45 - 12:00	1
9:55 - 11:30	2
11:00 - 11:50	3 ← Max
15:00 - 19:00	1
18:00 - 20:00	2

So, min 3 platforms needed

Approach 1

Ask the interviewer beforehand, whether the starting time array is sorted or not.

If not, then sort both the starting time and ending time in ascending order.

```
start = [120, 50, 550, 200, 700, 850]
end = [600, 550, 700, 500, 900, 1000]
```

Sort both arrays in ascending order

```
start = [50, 120, 300, 550, 700, 850]
end = [500, 550, 600, 700, 900, 1000]
```

Firstly we will find out the maximum no. of platforms required occupied together.

For that we will maintain a plat variable & max variable

Initially, at i = 0,

plat = 1 & max = 1

At i = 1, start < end, then we need another platform

plat = 2 & max = 2

At i = 2, start < end, then we need another platform

plat = 3 & max = 3

At i = 3, start = end, we need another platform

plat = 3 & max = 3

At i = 4, start = end, we don't need another platform

plat = 2 & max = 3

At i = 5, start = end, we don't need another platform

plat = 2 & max = 3

We compare end of previous time slot, if 2 time slots available, we check least one first.

TC - O(2n log n) + O(2n)

SC - O(1)

Code →

```
int i=1, j=0;
while(i<n && j<n){
    if(arr[i] >= dep[j]){
        i++;
        plat++;
    }
    else if(arr[i] <= dep[j]){
        j++;
        plat--;
    }
    ans=max(ans, plat);
```

Job Sequencing Problem →

You are given a set of N jobs where each job comes with a deadline and profit. The profit can only be earned upon completing the job within its deadline. Find the number of jobs done and the maximum profit that can be obtained. Each job takes a single unit of time and only one job can be performed at a time.

E.g. N=4, Jobs = {(1, 4, 20), (2, 1, 10), (3, 1, 40), (4, 1, 30)}

{Jobid, Deadline, Profit}

Find max profit.

Evaluating

We check job with least deadline & max profit

Profit = 40.

{3, 1, 40}

This takes one unit of time and job gets finished.

Now, we can only chose another job having deadline > 1.

So, only option is

{1, 4, 20} Profit = 20

Total profit = 60

o/p → 60

Eg 2 {(1, 2, 100), (2, 1, 19), (3, 2, 27), (4, 1, 25), (5, 1, 15)}

Step 1: Sort array in ascending order with increasing deadline

{(5, 1, 15), (2, 1, 19), (4, 1, 25), (3, 2, 27), (1, 2, 100)}

Deadline of few jobs was same, so we sorted in increasing profit.

So, we do job

(3, 2, 27) Profit = 27

Next we do job

(1, 2, 100) Profit = 100.

Total profit = 127

Approach

The main goal is to maximize profit

- If we do this problem greedily, the first thing which comes in mind is first complete the job with max profit.

Step 1:

- Sort the array in descending order of profit.

- We perform any given job on its last deadline, so we can do all other tasks before it also.

So, intuition is → Perform every job on last day, or near to deadline, so other jobs can also be done before it.

id	deadline	profit
6	2	80
3	6	70
4	6	65
2	5	60
5	4	25
8	2	22
1	4	20
7	2	10

As we can see the last deadline / max deadline = 6.

So, we have total 6 days to maximize profit

-1	-1	-1	-1	-1	-1
1	2	3	4	5	6

Make an array of size 6, initially set to -1, indicating no job has been performed.

Let's start with first job, i.e. job id = 6

It's deadline = 2, so we can either do it on day 1 or day 2,

For now, we will do it on day 2 as it is better to do a job near deadline.

Profit = 80

So,

-1	6	-1	-1	-1	-1
1	2	3	4	5	6

Now, job id = 3, deadline = 6, so we do schedule it for last day

Profit = 80 + 70

-1	6	-1	-1	-1	3

Now, job id = 4, deadline = 6, but on 6th day we are already performing a job, so it cannot be done on day 6.

As 6th day is occupied, we will check to its left which index job is not yet scheduled.

Profit = 80 + 70 + 65

-1	6	-1	-1	4	3

Now, job id = 2, deadline = 2, we schedule it on day 4, so

Profit = 80+70+65+60

-1	6	-1	2	4	3
1	2	3	4	5	6

Now, job id = 5, deadline = 4, we schedule it on day 3, so

-1	6	5	2	4	3

Profit += 25

Now, job id = 6, deadline = 2, schedule it on day 1,

6	6	5	2	4	3

Profit += 22

Now, job id = 4, deadline = 4, start checking from 4, all days in its left are occupied, so we cannot schedule it.

Now, job id = 7, it also cannot be scheduled

So, 6 jobs performed.

If job sequence is asked, we can return the above array.

TC - $O(N \log N) + O(N * n)$

SC - $O(n)$ n - max deadline

Fractional Knapsack Problem →

Weight of N items and their corresponding values are given. We have to put these items in a knapsack of weight W such that the total value obtained is maximized.

Note: We can either take the item as a whole or break it into smaller units.

Eg: N = 3, W = 50, values[] = {100, 60, 120}

weight[] = {20, 10, 30}

Total value maximized?

~~Chose most profit first i.e. 120, it has weight 30~~

~~So, now left weight = 30~~

20	80
20	100
10	60
50	240

Approach

Greedily we will look for items which have Value/Weight maximum.

Value[] = {60, 100, 120}

weight[] = {10, 20, 30}

Value/Weight = {6, 5, 4}

1) Sort the values w.r.t Value/Weight in descending order

2) Pick entire weight of first & second

20	100
10	60
50	

But at weight = 30, we cannot pick up 30, so we only pick a fraction of it.

Our knapsack weight left = 20

So, from 3rd item we pick 20 kg weight

20	80
20	100
10	60
50	240

$\frac{120}{3} = 4 \times 20 = 80$

Answer

$T(-O(N \log N) + O(N)$

$S(-O(1)$

Find minimum number of coins →

Minimum no. of coins and/or notes needed to make the change?

{1, 2, 5, 10, 20, 50, 100, 500, 1000}

Eg - V = 70

O/P → 2

Explanation: We need a 50Rs note and a 20 Rs note

Approach: Start from back of the array i.e. i = n-1 and check whether V <= arr[i], if not then keep decrementing i

For V = 70, our i = 5, i.e. arr[i] = 50 < 70

So, we do V = V - arr[i]

= 70 - 50 = 20 left

We don't decrement i at this point, so that we can check again whether 50 <= 20 or not.

Here, it is not, so we decrement i.

Now, at i = 4, arr[i] = 20

can be used,

Maintain a cnt variable to count number of coins.

TC - O(V) in worst case

SC - O(1)

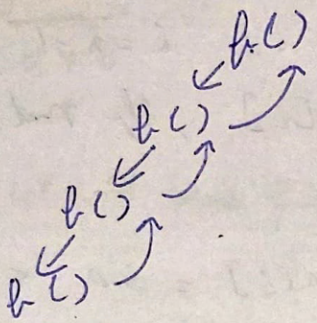
Recursion

When a function calls itself until a specified condition is met

Stack Overflow → Numerous function calls waiting due to recursion

Base condition → The condition at which recursion calls stops and functions gets popped from stack i.e. returns

Recursion Tree →



Multiple Recursion Calls →

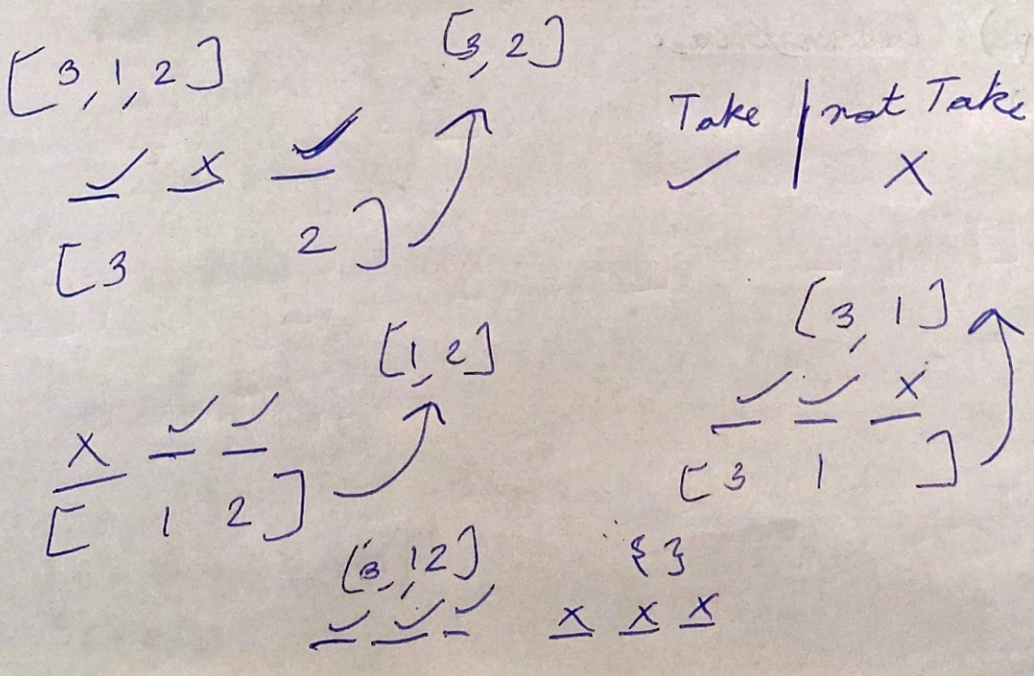
Print subsequence →

Subsequence → A contiguous/non contiguous sequence, which follows the order

arr → [3, 1, 2]

Subsequences → {}, 3, 1, 2, 31, 12, 32, 312

Total → 8



For every index we have two options, i.e. take it or don't take it

arr → [3, 1, 2]
0 1 2

```
f*n (index, arr[])
{
 if (index >= n)
    print (arr[])
    return;
[].add (arr[index]);
 f*n (index+1, []);        → Take
[].remove (arr[index]);
 f*n (index+1, []);        → Not Take
}
```

```
main ()
{
 arr → {3, 1, 2}
  f (0, [])
}
```

Let's see how it gets called →

f (0, arr[])

```
{
if (index >= n){   X No
}
```

[].add (3) → Now it contains first element i.e. 3

f (1, arr[]) → Take

```
f (1, [3])
{
 if () x
[3].add (arr[1]);   → 1 gets added
  f (2, arr[])
```

f(2, [3,1])

{ if() x

[3,1].add(arr[2]) → 2 gets added

f(3, ◻[]); → Take index 3

}

f(3, [3,1,2])

{ if() ✓

arr gets printed and returns to previous f(x-1)

}

f(2, [3,1])

{

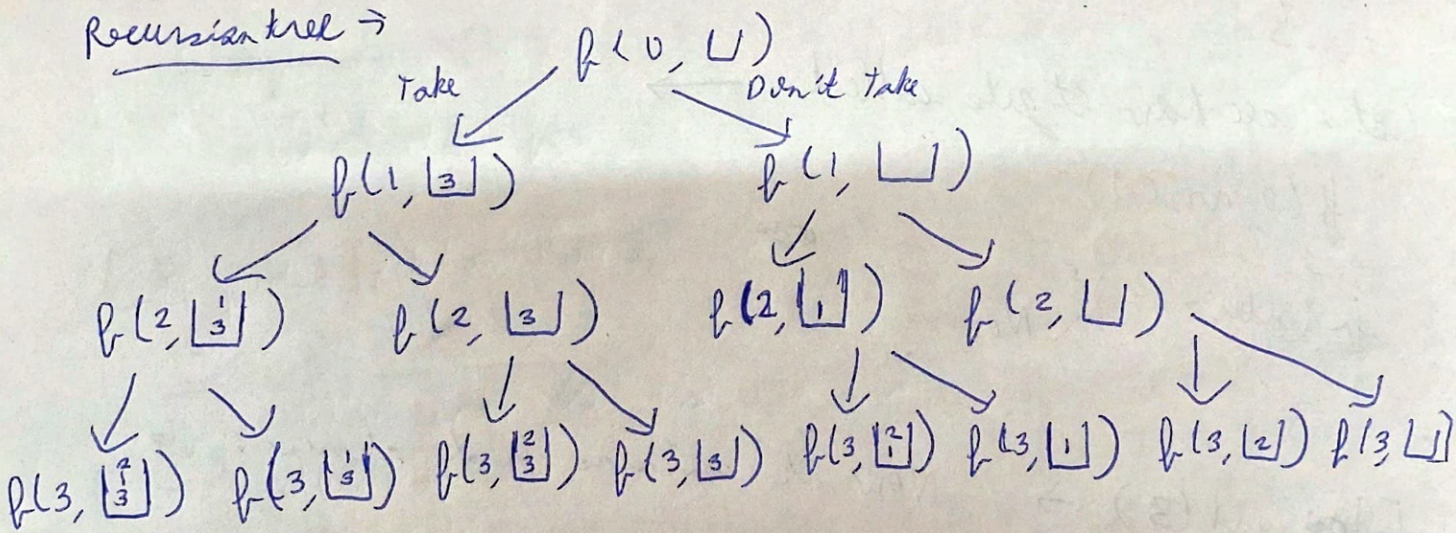
if() x

[3,1].remove(arr[2]); → Nahi kar rahe include 2 ka bta kyuki upar ise add kia tha

If, remove nahi krte, same print ho jata,

f(3, [3,1])

Recursion tree →



So, all subsequences are

{3,1,2} {3,1} {3,2} {3} {1,2} {1} {2} {}

Printing Subsequences whose Sum is K

(Take / not-take)

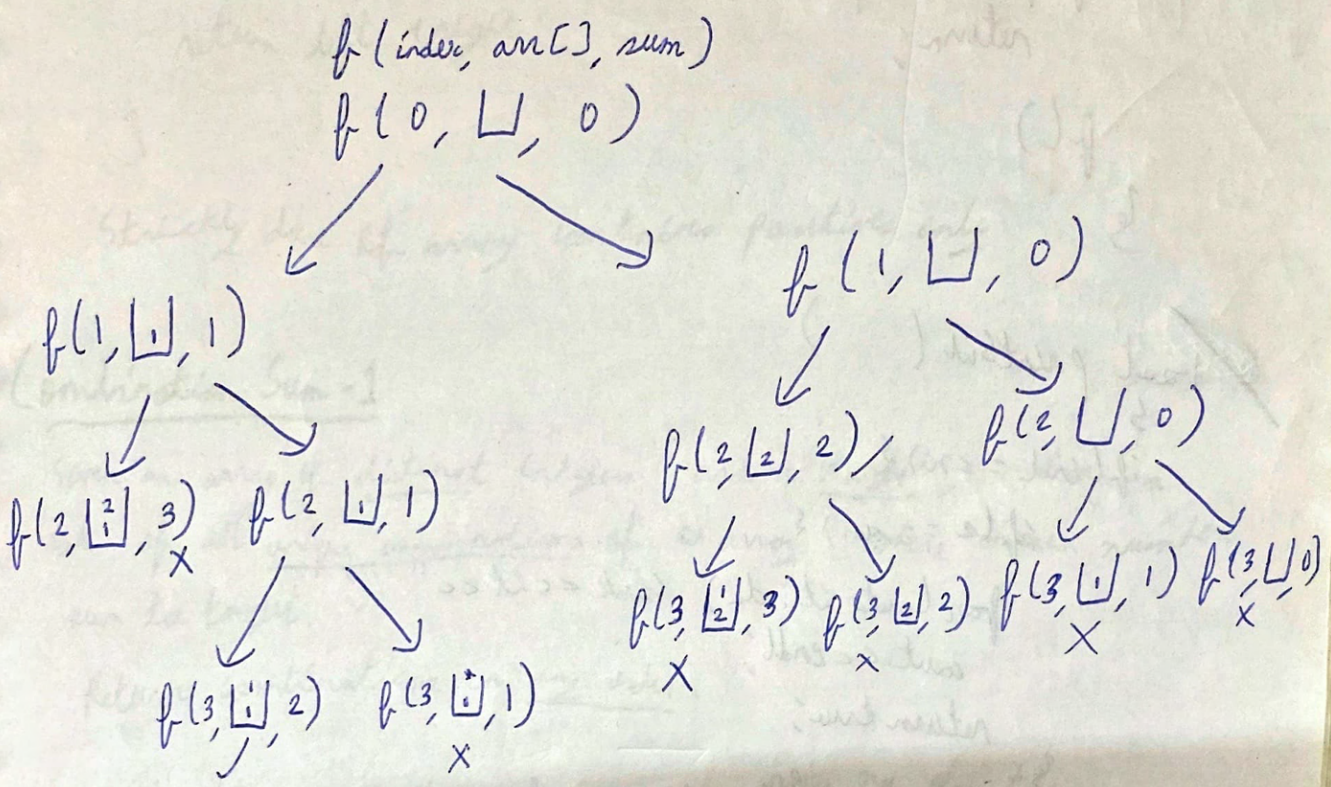
(V. IMP)

arr → [1, 2, 1] Sum = 2

Print subsequences whose (sum == 2)

O/P → [1,1], [2]

Approach → We will follow the same take / not take approach, here we need not maintain a separate data structure, we can only maintain a sum variable and add arr[i] to it, or not add arr[i] to it.



Wherever sum becomes > k, we return

& i >= n, we return.

And when (sum == k) we print the subsequence

```
if (i == n)
{
  if (s == sum)
    print(l)
  return;
}
```

But we want only one subsequence whose sum == k, so other function calls are unnecessary once the required subsequence is found. So, to * avoid further f^n calls.

** V.IMP

```
f(){
// base condn
   cond → satisfied
   return true
   else
   return false

if(f() == true)
   return;

f()
}
```

```
bool printSub( )
{
if(ind == n){
    if(s == sum){
      for(auto it: ds) cout<<it<<" ";
       cout<<endl;
       return true;
    }
    else return false;
}
ds.push_back(arr[ind]);
 s += arr[ind];
if(printS(ind+1, ds, s, sum, arr, n) == true){
   return true;
}

s -= arr[ind];
ds.pop_back();
if(printS(ind+1, ds, s, sum, arr, n) == true){
 return true; }
return false; }
```

Now, suppose if the question asks no. of subsequences, then

```
int f(){
    base case
    return 1  → satisfied
    return 0  → not satisfied
    @@
    left = f()
    right = f()
    return left + right;
}
```

**
V. IMP

Strictly done if array contains positives only

Combination Sum - 1

Given an array of distinct integers, and a target, return a list of all unique combinations of the array where chosen numbers sum to target.

Return combinations in any order.

- Same number may be chosen from array any no. of times.

E.g - arr = [2, 3, 6, 7], target = 7

O/P → [[2, 2, 3], [7]]

E.g 2 → [2, 3, 5] target = 8

[[2, 3, 3] [3, 5], [2, 2, 2, 2]]