* Subarray is a contiguous part of an array → ex: {2,3,5} → {2,3} ✓ , {2,5} ✗

Q) LONGEST SUBARRAY WITH GIVEN SUM K (+ & -)

ex: i/p: N=3, K=5, arr[] = {2,3,5}
o/p: 2

i/p: N=3, K=1, arr[] = {-1, 1, 1}
o/p: 3

A)
```
int getLongestSubarray(vector<int> &nums, int k){
    map<long long, int> preSumMap;
    long long sum = 0;
    int maxLen = 0;
    for(int i = 0; i<nums.size(); i++){
        sum += nums[i];
        if(sum == k) maxLen = max(maxLen, i+1);
        long long rem = sum - k;
        if(preSumMap.find(rem) != preSumMap.end()){
            int len = i - preSumMap[rem];
            maxLen = max(maxLen, len);
        }
        if(preSumMap.find(sum) == preSumMap.end()){
            preSumMap[sum] = i;
        }
    }
    return maxLen;
}
```

→ can use 'unordered_map' too.

* $T_c = O(N * \log N)$ (traversing array; time taken to find ele. in map)

to understand
ex: arr = [1,2,3,-1,1,1,-1,2,3], K=3

to understand
ex: arr = [2,0,0,3], K=3

Q) LONGEST SUBARRAY WITH GIVEN SUM 'K' (+ve & 0s)

A) * Prev. approach is also correct, but for this que., approach can be further optimised!

```
int longestSubarrayWithSumK(vector<int> a, long long k){
    int n = a.size();
    long long sum = a[0];
    int i=0, j=0, maxLen=0;
    while(j<n){
        while(sum>k and i<=j){
            sum -= a[i];
            i++;
        }
        if(sum==k){
            maxLen = max(maxLen, j-i+1);
        }
        j++;
        if(j<n) sum += a[j];  --> 1) we are on last ele. --> 2) j++ -> out of bounds
    }                              (that's why we are checking agar)
    return maxLen;
}
```

* $T_c = O(2*N)$

$S_c = O(1)$

BIT MANIPULATION

Convert decimal to binary: ex: 13 → (1101)$_2$ (remainder)

2 | 13	1
2 | 6	0
2 | 3	1
1	

&

binary to decimal: ex: $(1101)_2 \rightarrow 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 = 13$

Algo: (D→B) $T_c = (\log_2 n)$, $S_c = (\log_2 n)$

```
string convert2Binary(int n){
    res = "";
    while(n!=1){
        if(n%2==1) res += '1';
        else res += '0';
        n = n/2;
    }
    reverse(res);
    return res;
}
```

Algo: (B→D) $T_c = O(len)$, $S_c = O(1)$

```
int convert2Decimal(string x){
    int len = x.length, p2=1, num=0;
    for(i=len-1 -> 0){
        if(x[i]=='1') num = num + p2;
        p2 = p2*2;
    }
    return num;
}
```

1's compliment: ex: (13) = $(1101)_2$ —flip→ $(0010)_2$ } 1's comp. of 13

2's compliment → 1's compliment → Add '1' to it

ex: (13) → $(1101)_2$ —flip→ $(0010)_2$ —+1→ $0010 + 0001 = (0011)_2$

Operators (AND, OR, XOR, NOT, shift):

- AND → all true → true; 1 false → false
- OR → 1 true → true; all false → false
- XOR → no. of 1's → odd → 1; no. of 1's → even → 0

ex: x = (13 ^ 7) → 1101 ^ 0111 = (1010)$_2$ = 10

Note: XOR of same numbers = 0; num ^ 0 = num

- >> {Right Shift Operator}: ex: x = 13 >> 1 = 6

($\therefore x >> k = \frac{x}{2^k}$)

1101 → (shift right by 1) → 110 → 6

x = 13 >> 2 = 3

1101 (by 2) → 11 → 3

Largest int → 0 1 1 1 ... (last bit for sign) $2^{30}+2^{29}+\dots+2^0 = \frac{1}{2^0} = (2^{31}-1)$ → INT_MAX

Smallest int → 1 0 0 ... 0 ∴ (-2^{31}) → INT_MIN

(comp. stores -ve value in form of 2's compliment)

0 11 ... 1 {1's comp.} + 1 = 100 ... 0

- << {Left Shift Operator}: ex: 13 << 1

($\therefore num << k = num \times 2^k$)

00 1101 → $1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

1 1010 → $1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$

(Left shifted by 1 bit)

Overflow condn.: $(2^{31}-1) << 1$

- NOT (~)

1) Flip
2) Check (-)ve (yes → 2's comp.; No → Stop)

ex: x = ~(5) ⇒ 00 ... 101 → 11 ... 010 → this is (-)ve! → so convert to 2's comp (1's comp + 1) → 100 ... 0110

∴ Computer stores "-6"

ex: x = ~(-6) ⇒ 1 ... 11010 → (2's comp. of 6)

(comp. stores '-6' in 2's comp. form) ← flip 0 ... 00101

answer = "5"

Swap 2 numbers:

We can do it using 'xor' & without using 3rd variable

Algo:

a = a^b

b = a^b → (a^b)^b = a

a = a^b → (a^b)^a = b

Check if i^{th} bit is set or not: ($T_c = O(1)$, $S_c = O(1)$)

- Using <<: ex: N = 13, i = 2 → (1 1 0 1) & doing 1 << 2 → 100

  1101 & 100 = 00100

  ∴ Algo: if(N & (1<<i)) ✓ else ✗

Set → means bit value = '1'

- Using >>: ex: N = 13, i = 2 → (1101) → doing 1101 >> 2 = 0011 & 0001 = 0001 ('and' with '1')

  ∴ Algo: if((N>>i) & 1 == 0) ✗ else ✓

Set the i^{th} bit:

ex: N = 9, i = 2 → doing 1 << 2 → 00..0100

1001 (or) (|) 0100 = 1101

∴ Algo: (N | (1<<i)) — making i^{th} bit in N as '1'

Clear the i^{th} bit:

ex: N = 13, i = 2 → doing ~(1<<2) → ~(0100) → (1011)

1101 & 1011 = 1001

∴ Algo: (N & ~(1<<i)) — making i^{th} bit in N as '0'

Toggle the i^{th} bit:

Hint: XOR → ex: N = 13, i = 2: 1101 ^ 0100 = 1001

ex: N = 13, i = 1: 1101 ^ 0010 = 1111

∴ Algo: (N^(1<<i))

Remove the last set bit (rightmost):

ex: N = 40 → (101000)

N = 39 → (100111)

Soln: (remains same) (bit gets flipped)

∴ Algo: (N & (N-1)) → ex: 40 & 39 = 100000

Check if number is a power of 2:

ex: 32 → 100000 → every power of '2' will have only '1' setbit

ex: 32 → 100000 & 31 → 011111 = 000000

∴ Algo: if(N & (N-1) == 0) ✓ else ✗

Count number of set bits:

we know, last bit of a odd no. = '1' always

ex: N = 84 → 1010100 & N-1 = 83 → 1010011 = 1010000 (Turning off (1) this bit) → Again (N-1) → 1010000 & 1001111 = 1000000 → again (N-1) → 1000000 & 0111111 = 0000000

∴ Algo:
```
cnt = 0
while (N != 0){
    N = N&(N-1);
    cnt++;
}
```
T_c = O(No. of set) → worst case → O(31)

* In C++, we have built in func. to count no. of set bits → cnt = __builtin_popcount(n)

Minimum bit flips to convert no.:

ex: start = 10, goal = 7 ⇒ 1010 ^ 0111 (doing xor) = 1101 → ans → no. of set bits in ans

Algo:
```
ans = start ^ goal;
cnt = 0;
for(i=0 to 31){   // since 'int'
    if(ans & (1<<i)) cnt++;   // (checks the individual bit = 1)
}
```
Reduce T_c → ex: 13 divide by 2: 13 → 1, 6 → 0, 3 → 1, 1 → 1 → cnt = 3 → divide by '2', if (1) → cnt++, else → don't count

∴ $T_c = \log_2(start \wedge goal)$

$S_c = O(1)$

Q) Given an integer N, find the XOR of all no.s from 1 to N.

A) Brute force → for loop (1 to N)

Method 2:

N	(XOR)
N=1	1
N=2	3
N=3	0
N=4	4
N=5	1
N=6	7
N=7	0
N=8	8
N=9	1

→ pattern!

∴ Algo: $T_c = O(1)$, $S_c = O(1)$
```
func(N){
    if(N%4==1) return 1;
    else if(N%4==2) return N+1;
    else if(N%4==3) return 0;
    else return N;
}
```

* If we need XOR btw range (L–R)

ex: L = 4, R = 7 → 4^5^6^7 = ? → (1^2^3)^(1^2^...^6^7)

∴ Algo: $T_c = O(1)$, $S_c = O(1)$
```
f(L,R){
    f(L-1) ^ f(R);   → XOR(L-R)
}
```

Q) Divide 2 int. without using multiplication & division operators?

A) ex: dividend = 22, divisor = 3

$3\times2^0 = 3$
$3\times2^1 = 6$
$3\times2^2 = 12$
$3\times2^3 = 24$ ×

22 − 12 = 10 → $3\times2^0 = 3$, $3\times2^1 = 6$, $3\times2^2 = 12$ ×

10 − 6 = 4 → 4 − 3 = 1

(1 < 3) → stop!

Algo.:

```
fun(dividend, divisor){
    if(dividend == divisor) return 1;
    sign = true;
    if ((dividend >= 0 and divisor < 0) or (dividend < 0 and divisor > 0))   sign = false
    n = abs(dividend), d = abs(division), ans = 0;
    while (n >= d){                          --> log2 N
        count = 0;
        while (n >= (d << (count + 1))){     --> log2 N      (d x 2^(count+1))
            count ++;
        }
        ans += (1 << count);                 --> 2^count
        n = n - (d << count);                (directly does d x 2^count)
    }
    if (ans == (1<<31) and sign == true) return INT_MAX;
    if (ans == (1<<31) and sign == false) return INT_MIN;
    return sign ? ans : (-1 x ans);
}
```

$T_c = O(\log_2 N)^2$

Set the rightmost unset bit

Code:

```
int setBit(int N){
    int ans = (N&(N+1));      --> checking if power of 2!
    if(ans == 0) return N;
    else return (N|(N+1));
}
```

ex: n = 7 → 7

Power Set {Print all subsets}

ex: nums = [1, 2, 3], n = 3 → Total subsets = $2^n = 2^3 = 8$

	2	1	0	
0 →	0	0	0	→ [ ]
1 →	0	0	1	→ [1]
2 →	0	1	0	→ [2]
3 →	0	1	1	→ [1,2]
4 →	1	0	0	→ [3]
5 →	1	0	1	→ [1,3]
6 →	1	1	0	→ [2,3]
7 →	1	1	1	→ [1,2,3]

(Checking if the bit is set, then including it in our subset!)

Condn. ⇒ (N & (1<<i))

Code:

```
vector<vector<int>> subsets (vector<int> &nums){
    int n = nums.size(), subsets = 1 << n;      --> 2^n
    vector <vector <int>> ans;
    for(int num = 0; num < subsets; num++){
        vector<int> sub;
        for(int i = 0; i<n; i++){               --> N
            if(num & (1<<i)){
                sub.push_back(nums[i]);
            }
        }
        ans.push_back(sub);
    }
    return ans;
}
```

$T_c = N \times 2^n$

$S_c = 2^n \times N$ (total subsets) (size of each subset)

Q) All no.'s in array repeated thrice except 1 no. Find that no.?

A) **M1**: Use map & find freq = 1

M2: ex: nums = [5, 5, 5, 6, 4, 4, 4]

	(2)	(1)	(0)
5 →	1	0	1
5 →	1	0	1
5 →	1	0	1
6 →	1	1	0
4 →	1	0	0
4 →	1	0	0
4 →	1	0	0

'0' → has 3 bits set → 3 % 3 = 0

'1' has 1 bit set → 1 % 3 = 1 → so our ans will have 1st idx set

'2' has 7 bits set → 7 % 3 = 1

∴ ans = 110 = 6

Algo.: → $T_c = O(N \times 32)$, $S_c = O(1)$

```
for(bitIndex = 0 -> 31){
    cnt = 0;
    for(i=0 -> n-1){          (n = size of 'nums')
        if(nums[i] & (1 << bitIndex)){
            cnt++;
        }
    }
    if(cnt % 3 == 1){
        ans = ans | (1 << bitIndex);
    }
}
return ans;
```

M3: → $T_c = O(N)$, $S_c = O(1)$

Algo:

```
ones = 0, twos = 0;
for(i = 0 -> n-1){
    ones = (ones ^ nums[i]) & ~twos;
    twos = (twos ^ nums[i]) & ~ones;
}
return ones;
```

→ $T_c = O(N)$, $S_c = O(1)$

* nums[i] will go to 'ones', if not in 'twos'
* nums[i] will go to 'twos', if it is in 'ones'.
* 'ones' → stores ele. that occurs once, 'twos' → " " " twice!

Q) All no.'s are repeated twice in an array except 2 no.'s. Find those 2 no.'s?

A)

```
long numXor = 0;                          (given array)
for(auto it: nums) numXor ^= it;
long rightBit = numXor ^ (numXor & (numXor - 1));
int a = 0, b = 0;       --> 2 bucket integer which store no.'s based on this rightbit
for(auto it: nums){
    if(it & rightBit) a ^= it;     --> if rightBit is set!
    else b ^= it;
}
return {a, b};
```

Approach: As xor = 0, & unique 2 ele xor will contain a no. which contains bits which are not same in both. To distinguish no., we need to find that bit which is different in both. So, now finding rightmost set bit → based on this we can divide it into 2 set/bucket.

both buckets each will have 1 unique no. as it based on right most set bit.

{Dry run example for better understanding}

Q) KADANE'S ALGORITHM { Refer prob. before 'Bit Manipulation' topic }

Given integer array, find the subarray with largest sum & return it's sum. Also print that subarray.

A) Brute force: Use 2 loops for traversal → i=0 to n → sum=0 → j=i to n → sum += arr[j]. → maxi = max(maxi, sum)

$T_c = O(N)$, $S_c = O(1)$

Optimal: (for finding largest sum)

```
int maxSubArray(vector<int>&nums){
    int n = nums.size();
    int maxi = INT_MIN, sum = 0;
    for(int i=0; i<n; i++){
        sum += nums[i];
        if(sum > maxi) maxi = sum;
        if(sum < 0) sum = 0;
    }
    return maxi;
}
```

Optimal (for printing)

```
int maxSubArray(vector<int> &nums){
    int n = nums.size();
    int maxi = INT_MIN, sum = 0;
    int start = 0, ansStart = -1, ansEnd = -1;
    for(int i = 0; i<n; i++){
        if(sum == 0) start = i;   // max sum can't be found when sum becomes '0'.
        sum += nums[i];
        if(sum > maxi){
            maxi = sum;
            ansStart = start;
            ansEnd = i;
        }
        if(sum < 0) sum = 0;
    }
    cout << "The subarray is : [";
    for(int i = ansStart; i<=ansEnd; i++){
        cout << arr[i] << " ";
    }
    cout << "]" << endl;
    return maxi;
}
```

Q) Array → size 'N'. Without altering relative order of it's -ve & +ve, return an array of alternately positive & negative values.

Note: Start the array with (+)ve no.'s, array length = 'even'

A) Optimal

```
vector<int> rearrangeArray(vector<int> &nums){
    int n = nums.size(), positive = 0, negative = 1;
    vector<int> ans(n);
    for(int i=0; i<n; i++){
        if(nums[i] > 0){
            ans[positive] = nums[i];
            positive += 2;
        }
        else{
            ans[negative] = nums[i];
            negative += 2;
        }
    }
    return ans;
}
```

$T_c = O(N)$, $S_c = O(N)$

Q) LONGEST CONSECUTIVE SEQUENCE

Given an unsorted array → 'nums', return the length of longest consecutive elements sequence?

ex: i/p: [100,4,200,1,2,3] o/p: 4

i/p: [0,3,7,2,5,8,4,6,0,1] o/p: 9

A) Better appr: ($T_c = O(n\log n) + O(n)$)

```
int longestConsecutive(vector<int>&nums){
    if(a.size()==0) return 0;
    sort(a.begin(), a.end());
    int lastSmaller = INT_MIN, count = 0;
    int longest = 1;
    // find longest sequence
    for(int i=0; i<a.size(); i++){
        if(a[i] - 1 == lastSmaller){
            // a[i] = next ele. of current sequence
            count++;
            lastSmaller = a[i];
        }
        else if(a[i] != lastSmaller){
            count = 1;
            lastSmaller = a[i];
        }
        longest = max(longest, count);
    }
    return longest;
}
```

$T_c = O(N\log N) + O(N)$, $S_c = O(1)$

Optimal appr.:

```
int longestConsecutive(vector<int>&nums){
    if(nums.size()==0) return 0;
    unordered_set<int> st;
    for(int i=0; i<nums.size(); i++){
        st.insert(nums[i]);
    }
    int longest = 1;
    for(auto it:st){
        if(st.find(it-1) == st.end()){   // starting num from where consecutiveness starts!
            int count = 1, x = it;
            while(st.find(x+1) != st.end()){   // finding consecutive numbers
                x++; count++;
            }
            longest = max(longest, count);
        }
    }
    return longest;
}
```

$T_c = O(N) + O(2N)$, $S_c = O(N)$

unordered_set → find() → O(1) → Avg T_c; O(n) → worst T_c

Q) SUBARRAY SUM EQUALS K

Given array → 'nums' & integer 'k', return total no. of subarrays whose sum = k.

ex: N=4, arr = {3,1,2,4,9}, k=6 → o/p: 2

A) Better appr.:

```
count = 0, n = nums.size();
for(i=0 to n){
    ans = 0;
    for(j=i to n){
        ans += nums[j];
        if(ans == k) count++;
    }
}
return count;
```

Optimal appr.: {prefix sum + maps}

ex: arr = [1,2,3,-3,1,1,1,4,2,-3], k=3

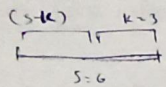
- We declare map to store prefix sum & their counts
- Set value of 0 as 1 on map.
- Loop (i=0 to n), we will do:
  - add curr. ele. arr[i] to prefix sum
  - calculate prefix sum i.e. 'x-k' for which we need the occurrence (if prefixSum > k, then we find occurrence of 'x-k' in map)
  - we will add the occurrence of prefix sum 'x-k', 'm-k' i.e. mp[x-k] to our answer
  - then we will store the current prefix sum in the map ↑ it's occurrence by 1.

(Dry run in next page)

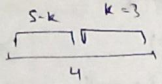
arr = [1, 2, 3, -3, 1, 1, 1, 4, 2, -3] , k = 3

presum = 0 1 3 6 3 4 5 6 10 12 9

count = 0 1 2 2 3 4 5 6



6-3=3



4-3=1

(already in map → increase count)

(presum, count)

(9,1)
(12,1)
(10,1)
(5,1)
(4,1)
(6,1)2
(3,1)2
(1,1)
(0,1)

(So we remove those etc.'s where pref. sum ≠ k-3)

: [1, 2, 3, -3, 1]

0 1 3 6 3 4

```
vector<vector<int>> subArraySumK(vector<int>& nums, int k){
    unordered_map<int, vector<int>> mp;
    mp[0] = {-1};
    int presum = 0;
    vector<vector<int>> result;
    for(int i=0; i<nums.size(); i++){
        presum += nums[i];
        int remove = presum - k;
        if(mp.find(remove) != mp.end()){
            for(int index : mp[remove]){
                result.push_back(vector<int>(nums.begin()+index+1,
                                             nums.begin()+i+1));
            }
        }
        mp[presum].push_back(i);
    }
    return result;
}
```

{print 2D array in 'int main'}

FIND FIRST & LAST OCCURENCE OF A GIVEN NUM. IN A SORTED ARRAY

```
vector<int> searchRange(vector<int>& nums, int target){
    vector<int> ans(2, -1);
    if(nums.empty()) return ans;
    int left=0, right=nums.size()-1;
    while(left<=right){                    // BS to find starting position of target
        int mid = (left+right)/2;
        if(nums[mid] >= target){
            if(nums[mid]==target) ans[0]=mid;
            right = mid-1;
        }
        else{
            left = mid+1;
        }
    }
    left=0; right=nums.size()-1;           // reset to find end position of target
    while(left<=right){                    // BS to find ending position of target
        int mid = (left+right)/2;
        if(nums[mid] <= target){
            if(nums[mid]==target) ans[1]=mid;
            left = mid+1;
        }
        else{
            right = mid-1;
        }
    }
    return ans;
}
```

PRIME FACTORS OF A NUMBER

A) App1: for(i=1→√N) → if(N%i==0) → if(isPrime(i)) → (add)
↳ if(n/i != i) → if(isPrime(n/i)) → (add)

↳ $T_c = O(\sqrt{N} \times 2 \times \sqrt{N})$

App2: for(i=2→√N) → if(N%i==0) → (add)
↳ while(n%i==0) → n = n/i

ex: N = 780

2	780
2	390
3	195
5	65
13	13
	1

Code: (above one correct, but 1 case → if N = prime no.)

```
vector<int> ans;
for(int i=2; i<=sqrt(n); i++){
    if(n%i==0){
        ans.push_back(n);
        while(n%i==0) n=n/i;
    }
}
if(n!=1) ans.push_back(n);
return ans;
```

$T_c = O(\sqrt{N})$

$S_c = O(\sqrt{N})$

ex: N = 37 → i = 2→6; n = n

↳ n != 1 → 37 = prime

PRIME FACTORS TILL NUMBER N → {SIEVE OF ERATOSTHENES}

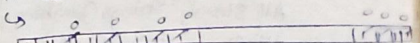
A)
```
int countPrimes(int n){ → Tc = O(N log(log N))
    int prime[n+1];
    //precomputation
    for(int i=2; i<=n; i++) prime[i]=1;
    for(int i=2; i<=sqrt(n); i++){
        if(prime[i]==1){
            for(int j=i*i; j<=n; j+=i){
                prime[j]=0;
            }
        }
    }
    int count=0;
    for(int i=2; i<n; i++){
        if(prime[i]==1) count++;
    }
    return count;
}
```

↳ refer how it came

$S_c = O(n)$

explanation:

n = 30



2×2	3×2	5×2		
2×3	3×3	5×3		
2×4	3×4	5×4	7×6	
2×5	3×5	5×5	7×7	
			7×8	

prev. all already mentioned in others prod.

CHECK IF ONE STRING IS ROTATION OF ANOTHER STRING

ex: abac, baca → true

```
bool isCyclicRotation(string &s, string &goal){
    if(s.size() != goal.size()) return false;
    string doubleS = s+s;
    return doubleS.find(goal) != string::npos;
}
```

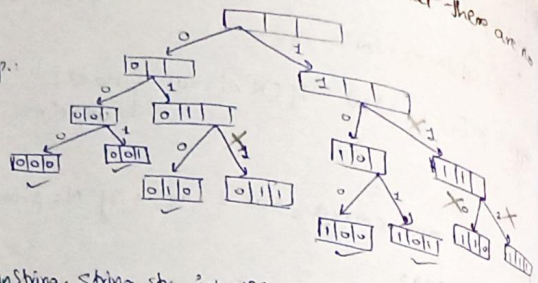
{if substring not found, then 'find()' returns 'string::npos' to indicate that the substring was not found.}

Q) BINARY STRINGS WITH NO CONSECUTIVE 1's

Given int N, generate all binary strings of length 'N' such that there are no consecutive 1's in string.

ex: N = 3

o/p: 000 001 010 100 101 } exp:



Code:

```
void All_Binary_Strings(vector<string>& binString, string str, int N){
    int len = str.size();
    if(len == N){
        binString.push_back(str);
        return;
    }
    else if(str[len-1] == '1'){ ⟶ if prev = '1', then present char can be '0' only.
        All_Binary_Strings(binString, str+'0', N);
    }
    else if(str[len-1] == '0'){ → if prev = '0', then present char can be either '0' or '1'.
        All_Binary_Strings(binString, str+'0', N);
        All_Binary_Strings(binString, str+'1', N);
    }
}

vector<string> generateString(int N){
    vector<string> binaryStrings;
    string word;
    word.push_back('0');
    All_Binary_Strings(binaryStrings, word, N);
    word[0] = '1';
    All_Binary_Strings(binaryStrings, word, N);
    return binaryStrings;
}
```

Q) GENERATE PARENTHESES ; ex: n=3 , o/p: ["((()))", "(()())", "(())()", "()(())", "()()()"]

```
A) void all_Parentheses(vector<string>& paren, string str, int n, int front, int back){
    int len = str.size();
    if(len == n*2){
        paren.push_back(str);
        return;
    }
    else{
        if(front > back) all_Parentheses(paren, str+')', n, front, back+1);
        if(front < n) all_Parentheses(paren, str+'(', n, front+1, back);
    }
}

vector<string> generateParenthesis(int n){
    vector<string> parentheses;
    string word; word.push_back('(');
    all_Parentheses(parentheses, word, n, 1, 0);
    return parentheses;
}
```

{Draw recursion tree for practice}

maintain count of '('

maintain count of ')'

Q) SUBARRAYS WITH SUM K ; generate & return all subarrays of array 'A', whose sum = 'K'.

(all de.'s in array = +')

ex: N = 6, K = 3; A = [1,2,3,1,1,1] → o/p: [1,2], [3], [1,1,1]

```
A) void checkSum(vector<int>& count, vector<int>& a, long long k, long long index, long long sum,
        vector<vector<int>>& subArrays){
    if(sum == k) subArrays.push_back(count);
    if(sum > k){
        while(sum > k and !count.empty()){
            sum -= count[0];
            count.erase(count.begin());
        }
        if(sum == k) subArrays.push_back(count);
    }
    if(index == a.size()) return;
    count.push_back(a[index]);
    checkSum(count, a, k, index+1, sum+a[index], subArrays);
}

vector<vector<int>> subArraysWithSumK(vector<int> a, long long k){
    long long sum = 0, index = 0;
    vector<int> count;
    vector<vector<int>> subArrays; → to store our answers
    checkSum(count, a, k, index, sum, subArrays);
    return subArrays;
}
```

Check Hashing approach ↓ prev.

Q) PRINT ALL SUBSEQUENCES ~ POWER SET → input = string

A) M1: Using bit manipulation (written before) ⟶ $T_c = O(2^n * n)$, $S_c = O(n)$

M2: Using backtracking → $T_c = O(2^n)$, $S_c = O(n)$

```
vector<string> solve(int index, string s, string temp, vector<string>& ans){
    if(index == s.length()){
        ans.push_back(temp);
        return ans;
    }
    temp = temp + s[index];
    solve(index+1, s, temp, ans); → take
    temp.pop_back();
    solve(index+1, s, temp, ans); → not take
    return ans;
}

vector<string> generateSubsequence(string s){
    string temp = "";
    vector<string> ans;
    return solve(0, s, temp, ans);
}
```

index

Q) PRINT ALL UNIQUE SUBSETS → i/p: no.

A) (1) Generating all subsets & storing it in ⇒ set<vector<int>>

$T_c = O(2^n \cdot (k \log(x)))$

(2^n: for generating all subsets; $k \log(x)$: insert every comb. of avg. length 'k' in a set of size 'x')

(2)

```
void subSum(int index, vector<int>&nums, vector<int>&temp,
                                  vector<vector<int>>&ans){
    ans.push_back(temp);
    for(int i=index; i<nums.size(); i++){
        if(i!=index and nums[i]==nums[i-1]) continue;
        temp.push_back(nums[i]);
        subSum(i+1, nums, temp, ans);
        temp.pop_back();
    }
}
vector<vector<int>> subsetsWithDup(vector<int>& nums){
    vector<vector<int>> answer;
    vector<int> temp;
    sort(nums.begin(), nums.end());
    subSum(0, nums, temp, answer);
    return answer;
}
```

$T_c = O(2^n \cdot k)$ (store every subset of avg. length k)

$S_c = O(2^n \cdot k)$ → store every subset of avg. length k.

Q) COUNT WITH 'K' DIFFERENT CHARACTERS

ex: s="abcad", k=2 → o/p: 4 → reason: {ab, bc, ca, ad}

ex: s="aacfssa", k=3 → o/p: 5 → reason {aacf, acf, cfs, cfss, fss}

```
A) int countSubStrings(string str, int k){
    int answer=0, i=0;
    while(i<str.size()){
        int count=0;
        bool visited[26]={false};
        int j=i;
        while(j<str.size()){
            if(visited[str[j]-'a']==false){
                visited[str[j]-'a']=true;
                count++;
            }
            if(count==k) answer++;
            if(count>k) break;
            j++;
        }
        i++;
    }
    return answer;
}
```

$T_c = O(n^2)$

$S_c = O(26)$

→ {condn. imp. for identification of → ex: "cfs", "cfss" in above ex.}

Q) SUM OF BEAUTY OF ALL SUBSTRINGS

beauty of string = difference btw frequency of most frequently occuring character & least frequently occuring character. Return sum of beauty of all substrings of 's'?

```
A) int sumOfBeauty(string s){
    int result=0;
    for(int i=0; i<s.size(); i++){
        vector<int> res(30,0);
        for(int j=i; j<s.size(); j++){
            int mx=INT_MIN, mn=INT_MAX;
            res[s[j]-'a']++;
            for(auto x:res){
                if(x!=0){
                    mn=min(mn,x);
                    mx=max(mx,x);
                }
            }
            result+=mx-mn;
        }
    }
    return result;
}
```

$T_c = O(n^2)$

$S_c =$

Q) FIND MINIMUM IN ROTATED SORTED ARRAY ; ex: [3,4,5,1,2] → o/p: 1

```
A) int findMin(vector<int>& nums){
    int lo=0, hi=nums.size()-1;
    while(lo<hi){
        int mid=(lo+hi)/2;
        if(nums[mid]<nums[hi]) hi=mid;   // right half → sorted, ∴ min. at left half or mid
        else lo=mid+1;                   // right half → not sorted, ∴ min. at right half
    }
    return nums[lo];
}
```

Q) FIND A PEAK ELEMENT II ; ex:

10	20	15
21	30	14
7	16	32

→ o/p: [1,1] or [2,2]

Peak ele. = greater than all of its adjacent neighbours to the left, right, top, & bottom

```
A) int findMaxIndex(vector<vector<int>>& mat, int n, int m, int col){
    int maxVal=-1, index=-1;
    for(int i=0; i<n; i++){
        if(mat[i][col]>maxVal){
            maxVal=mat[i][col];
            index=i;
        }
    } //return index of ele. which is greatest in that col.
    return index;
}
```

$T_c = O(m \log n)$

```
vector<int> findPeakGrid(vector<vector<int>>& mat){
    int n = mat.size(), m = mat[0].size();
    int lo = 0, hi = m-1;
    while(lo <= hi){
        int mid = (lo+hi)/2;
        int maxRowIndex = findMaxIndex(mat, n, m, mid);
        int left = mid-1 >= 0 ? mat[maxRowIndex][mid-1] : -1;
        int right = mid+1 < m ? mat[maxRowIndex][mid+1] : -1;
        if(mat[maxRowIndex][mid] > left and mat[maxRowIndex][mid] > right){
            return {maxRowIndex, mid};
        }
        else if(mat[maxRowIndex][mid] < left) hi = mid-1;
        else lo = mid+1;
    }
    return {-1, -1};
}
```

Q) MEDIAN IN A ROW-WISE SORTED MATRIX:

Given row-wise sorted matrix 'mat' → 'm×n' (rows, cols) → m,n = odd always. return median of matrix

ex: 1 3 8 → op: 3
2 3 4
1 2 5

```
A) int median(vector<vector<int>>& matrix, int m, int n){
    int lo = INT_MAX, hi = INT_MIN;
    for(int i=0; i<m; i++){
        lo = min(lo, matrix[i][0]);
        hi = max(hi, matrix[i][n-1]);
    }
    int ans;
    while(lo <= hi){
        int mid = (lo+hi)/2;
        int freq = 0;
        for(int i=0; i<m; i++){          // returns how many no.'s <= mid
            freq += upper_bound(matrix[i].begin(), matrix[i].end(), mid) - matrix[i].begin();
        }
        if(freq >= (m*n+1)/2){
            ans = mid;
            hi = mid-1;
        }
        else{
            lo = mid+1;
        }
    }
    return ans;
}
```

$T_c = (\log_2(10^9)) \times (n \times \log_2 m)$ — inside for loop; time taken by 'upperbound' fun.; no. of rows

search space → 'lo' can be as min. as 1, 'hi' can be as max as 10^9

Q) SORT 0's 1's 2's IN AN ARRAY

```
A) void sort012(int *arr, int n){
    int low = 0, mid = 0, high = n-1;
    while(mid <= high){
        if(arr[mid] == 0){
            swap(arr[mid], arr[low]);
            mid++; low++;
        }
        else if(arr[mid] == 1){
            mid++;
        }
        else{
            swap(arr[mid], arr[high]);
            high--;
        }
    }
```

Considering:
- 0 to low-1 → all 0's
- low to mid-1 → all 1's
- mid to high → unsorted
- high+1 to n-1 → all 2's

$T_c = O(N)$
$S_c = O(1)$

Q) ROTATE ELE's ONCE IN MATRIX:

given 'n×m' matrix, rotate ele.s clockwise

ex:
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

→

5	1	2	3
9	10	6	4
13	11	7	8
14	15	16	12

```
A) void rotateMatrix(vector<vector<int>>& mat, int n, int m){
    int t=0, b=mat.size()-1, l=0, r=mat[0].size()-1;
    while(t<b and l<r){
        int prev = mat[t+1][l];
        for(int i=l; i<=r; i++) swap(mat[t][i], prev);
        t++;
        for(int i=t; i<=b; i++) swap(mat[i][r], prev);
        r--;
        for(int i=r; i>=l; i--) swap(mat[b][i], prev);
        b--;
        for(int i=b; i>=t; i--) swap(mat[i][l], prev);
        l++;
    }
}
```

Q) MERGE 2 SORTED ARRAYS WITHOUT EXTRA SPACE:

ex: A={1,4,5,7}, B={2,3,6}
o/p: {1,2,3,4,5,6,7}

```
A) void merge(vector<int>& a, vector<int>& b){
    int n = a.size(), m = b.size();
    int len = n+m;
    int gap = len/2 + len%2;          // can use 'ceil' fun. too
    while(gap > 0){
        int left = 0;
        int right = left+gap;
        while(right < len){
            if(left < n and right >= n){
                if(a[left] > b[right-n]){
                    swap(a[left], b[right-n]);
                }
            }
            else if(left >= n){
                if(b[left-n] > b[right-n]){
                    swap(b[left-n], b[right-n]);
                }
            }
            else{
                if(a[left] > a[right]){
                    swap(a[left], a[right]);
                }
            }
            left++;
            right++;
        }
        if(gap == 1) break;
        gap = gap/2 + gap%2;
    }
}
```

{based on 'shell sorting'}

$T_c = (n+m)\log(n+m)$

ex: A = {1,2,3,4}
B = {5,6,7}

Q) MERGE ALL OVERLAPPING INTERVALS → ex: [1,3],[2,6],[8,10] → [1,6],[8,10]

A) vector<vector<int>> mergeOverlapInterval(vector<vector<int>>& arr)

```
{
  sort(arr.begin(), arr.end());
  vector<vector<int>> ans;
  for(int i=0; i<arr.size(); i++){
    if(ans.size()==0 or arr[i][0] > ans.back()[1]){
      ans.push_back(arr[i]);
    }
    else  ans.back()[1] = max(ans.back()[1], arr[i][1]);
  }
  return ans;
}
```

$T_c = O(N \log N) + O(N)$

$S_c = O(N)$

ans.back() → returns last element in 'ans'

Q) FIND DUPLICATE IN ARRAY : arr has 'n+1' integer where range = [1,n]

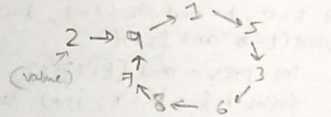
A) M1: Use set M2: Use linked list → if duplicate → cycle present.

```
int findDuplicate(vector<int>& nums){
  int slow = nums[0], fast = nums[0];
  do{
    slow = nums[slow];
    fast = nums[nums[fast]];
  }while(slow != fast);
  fast = nums[0];
  while(slow != fast){
    slow = nums[slow];
    fast = nums[fast];
  }
  return slow;
}
```

$T_c = O(N)$, $S_c = O(1)$

ex:

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



Q) ADD 2 NO.'s: ip: 2 linked lists with digits in reverse order → add 2 no. & return their sum → ex: (2)→(4)→(3) + (5)→(6)→(4) = (7)→(0)→(8); exp: 342 + 465 = 807

A)

```
Node* addTwoNums(Node* l1, Node* l2){
  Node* dummy = new Node();
  Node* temp = dummy;
  int carry = 0;
  while(l1!=NULL or l2!=NULL or carry!=0){
    int sum = 0;
    if(l1!=NULL){
      sum += l1->val;
      l1 = l1->next;
    }
    if(l2!=NULL){
      sum += l2->val;
      l2 = l2->next;
    }
    sum += carry;
    carry = sum/10;
    Node* temp1 = new Node(sum%10);
    temp->next = temp1;
    temp = temp->next;
  }
  return dummy->next;
}
```

$T_c = O(\max(m,n))$

$S_c = O(\max(m,n))$

Q) COUNT INVERSIONS : arr → all distinct values, find total inversions

inversion → is a pair (a[i], a[j]) → a[i] > a[j] , (i,j) ∈ [0,N) , i<j

ex: a = 2 5 1 3 4

op: 4 exp: (2,1),(5,1),(5,3),(5,4)

A)

```
int merge(vector<int>& arr, int low, int mid, int high){
  vector<int> temp;
  int left = low, right = mid+1;
  int cnt = 0, // to count pairs
  while(left<=mid and right<=high){
    // storing ele.'s in temporary array in sorted way
    if(arr[left] <= arr[right]){
      temp.push_back(arr[left]);
      left++;
    }
    else{
      temp.push_back(arr[right]);
      cnt += (mid-left+1); //Modification
      right++;
    }
  }
  while(left<=mid) temp.push_back(arr[left++]);
  while(right<=high) temp.push_back(arr[right++]);
  for(int i=low; i<=high; i++){
    arr[i] = temp[i-low];
  }
  return cnt; //Modification
}
```

starting index of left arr

starting index of right arr

{Optimal Approach}

```
int mergeSort(vector<int>& arr, int low,
              int high){
  int cnt=0;
  if(low>=high) return cnt;
  int mid = (low+high)/2;
  cnt += mergeSort(arr, low, mid);
  cnt += mergeSort(arr, mid+1, high);
  cnt += merge(arr, low, mid, high);
  return cnt;
}

int totalInversions(vector<int>& a, int n){
  return mergeSort(a, 0, n-1);
}
```

count no. of pairs

$T_c = O(N \log N)$

$S_c = O(N)$

Q) FIND MAJORITY ELE. THAT OCCURS MORE THAN N/2 (or) N/3 TIMES:

A) Moore's Voting Algorithm → (N/2): ans will have only 1 ele., (N/3): ans will have only 2 ele.'s

{N/2}:

```
int majorityEle(vector<int> v){
  int count = 0, ele = 0;
  for(int i=0; i<v.size(); i++){
    if(count==0){
      count = 1;
      ele = v[i];
    }
    else if(ele==v[i]) count++;
    else count--;
  }
  int count1 = 0; // checking if stored ele. is majority ele.
  for(int i=0; i<n; i++){
    if(v[i]==ele) count1++;
  }
  if(count1 > (n/2)) return ele;
  return -1;
}
```

applying algo.

$T_c = O(N)$

$S_c = O(1)$

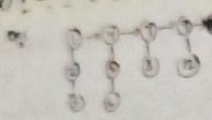
similar

```
vector<int> majorityEle(vector<int> v){
  int n=v.size(), cnt1=0, cnt2=0, el1=INT_MIN,
      el2=INT_MIN;
  for(int i=0; i<n; i++){
    if(cnt1==0 and el2!=v[i]){
      cnt1=1;
      el1=v[i];
    }
    else if(cnt2==0 and el1!=v[i]){
      cnt2=1;
      el2=v[i];
    }
    else if(v[i]==el1) cnt1++;
    else if(v[i]==el2) cnt2++;
    else{
      cnt1--;
      cnt2--;
    }
  }
  vector<int> ans; cnt1=0, cnt2=0, mini=n/3;
  for(auto i:v){
    if(i==el1) cnt1++;
    if(i==el2) cnt2++;
  }
  if(cnt1>=mini) ans.push_back(el1);
  if(cnt2>mini) ans.push_back(el2);
  return ans;
}
```

manually check if stored ele. in el1 & el2 are the majority ele.'s

Q) FLATTEN A LINKED LIST : Given LL of 'n' head nodes, where every node in LL has 2 pointers:
- 'next' : points to next node in list
- 'child' : points to a LL where curr_node is head

Return a single sorted layer.



```
A) Node* flattenLinkedList(Node* head){
    if(head==NULL || head->next==NULL) return head;
    head->next = flattenLinkedList(head->next);
    head = merge2LL(head, head->next);
    return head;
}
```

```
Node* merge2LL(Node* a, Node* b){
    Node* temp = new Node(0);
    Node* res = temp;
    while(a!=NULL and b!=NULL){
        if(a->data < b->data){
            temp->child = a;
            temp = temp->child;
            a = a->child;
        }else{
            temp->child = b;
            temp = temp->child;
            b = b->child;
        }
    }
    while(b!=NULL){
        temp->child = b;
        temp = temp->child;
        b = b->child;
    }
    while(a!=NULL){
        temp->child = a;
        temp = temp->child;
        a = a->child;
    }
    return res->child;
}
```

Q) FIND Nth ROOT OF M

e: n=3, m=27 → o/p: 3

```
A) int NthRoot(int n, int m){
    long long lo=1, hi=m, ans=-1;
    while(lo<=hi){
        long long x=-1, mid=(lo+hi)/2;
        for(long long i=1; i<=n; i++){
            x = x*mid;
            if(x>m) break;
        }
        if(x==m){ans=mid; break;}
        else if(x<m) lo=mid+1;
        else hi=mid-1;
    }
    return int(ans);
}
```

Q) SEARCH IN ROTATED SORTED ARRAY : e: 3,4,5,6,0,1,2 ; tgt=0 → o/p

```
A) int search(vector<int>& nums, int lo, int hi, int tgt){
    while(lo<=hi){
        int mid = (lo+hi)/2;
        if(nums[mid]==tgt) return mid;
        if(nums[lo]<=nums[mid]){ // left part is sorted
            if(nums[lo]<=tgt and tgt<=nums[mid]) hi=mid-1;
            else lo=mid+1;
        }
        else{ // right part is sorted
            if(nums[mid]<=tgt and tgt<=nums[hi]) lo=mid+1;
            else hi=mid-1;
        }
    }
    return -1;
}
```

Q) MEDIAN OF 2 SORTED ARRAYS eg: a1 → [1,3,4], a2 → [2,7] } o/p: 3.000

```
A) double median(vector<int>& a, vector<int>& b){
    int n1=a.size(), n2=b.size();
    if(n1>n2) return median(b,a);
    int n = n1+n2; // total length
    int left = (n1+n2+1)/2; // length of left half
    int lo=0, hi=n1;
    while(lo<=hi){
        int cut1 = (lo+hi)/2;
        int cut2 = ((n1+n2)/2) - cut1;
        int l1 = cut1==0 ? INT_MIN : arr1[cut1-1];
        int l2 = cut2==0 ? INT_MIN : arr2[cut2-1];
        int r1 = cut1==n1 ? INT_MAX : arr1[cut1];
        int r2 = cut2==n2 ? INT_MAX : arr2[cut2];
        if(l1>r2){
            hi = cut1-1;
        }
        else if(l2>r1){ lo = cut1+1; }
        else{
            return (n1+n2)%2==0 ? (max(l1,l2)+min(r1,r2))/2.0
                                : max(l1,l2);
        }
    }
    return 0;
}
```

Algo.: lo=0, hi=n1; while(lo<=hi){ cut1 = (lo+hi)/2; cut2 = ((n1+n2)/2) - cut1; l1 = cut1==0 ? INT_MIN : arr1[cut1-1]; l2 = cut2==0 ? INT_MIN : arr2[cut2-1]; r1 = cut1==n1 ? INT_MAX : arr1[cut1]; r2 = cut2==n2 ? INT_MAX : arr2[cut2]; if(l1>r2){ hi = cut1-1; } else if(l2>r1){ lo = cut1+1; } else{ return (n1+n2)%2==0 ? (max(l1,l2)+min(r1,r2))/2 : max(l1,l2); } }

e: 1 5 8 10 18 20
2 3 6 7

1 2 3 5 6 7 8 10 18 20

Divide into 2 parts:

1 5 | 8 10 18 20
2 3 6 | 7

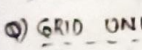
((n1+n2)/2) ((n1+n2)/2)

new median = max(l1,l2) + min(r1,r2) / 2

new partition of array must happen in such a way that: $l_1 < r_2$ & $l_2 < r_1$
{ $l_1 < r_1$ & $l_2 < r_2$ → already by default in array }

else lo = mid1 + 1;

return 0;

Q) GRID UNIQUE PATHS :  → at (0,0) → need to reach (m,n). How many paths.
- all blocks are empty so we can move.

A) Soln. 2 : Use DP {refer notes}

Soln. 3 :

```
int uniquePaths(int m, int n){
    int N = n+m-2;
    int r = m-1;
    double res = 1;
    for(int i=1; i<=r; i++) res = res * (N-r+i)/i;
    return (int)res;
}
```

Q) REVERSE PAIRS : Return no. of reverse pairs in array
↳ Pair(i,j) → 0 <= i < j < arr.length() ↳ nums[i] > 2 * nums[j]

```
A) int countPairs(vector<int>& arr, int lo, int mid, int hi){
    int right = mid+1, cnt = 0;
    for(int i=lo; i<=mid; i++){
        while(right<=high and arr[i]>(long long)2*arr[right]){
            right++;
        }
        cnt = cnt + (right - (mid+1));
    }
    return cnt;
}
```

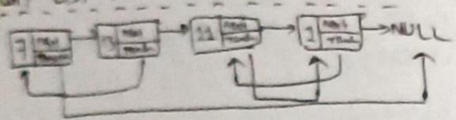
```
int mergeSort(vector<int>&arr, int low, int high){
    int cnt = 0;
    if(low >= high) return cnt;
    int mid = (low+high)/2;
    cnt += mergeSort(arr, low, mid);
    cnt += mergeSort(arr, mid+1, high);
    cnt += countPairs(arr, low, mid, high);
    merge(arr, low, mid, high); → see last page of Book (1)
    return cnt;
}
```

```
int reversePairs(vector<int>& nums){
    int n = nums.size();
    return mergeSort(nums, 0, n-1);
}
```

$T_c = O(2N \log N)$

$S_c = O(N)$

Q) COPY LIST WITH RANDOM POINTER:



Construct deep copy of list → (means create new list with exactly same configuration as given.)

We have 2 pointers: next, random

Algo.: → Insert copy nodes in between
→ Connect Random Pointers
→ Connect Next Pointers

```
Node* insertCopyBetweenHead(Node* head){
    Node* temp = head;
    while(temp != NULL){
        Node* copy = new Node(temp->val);
        copy->next = temp->next;
        temp->next = copy;
        temp = temp->next->next;
    }
    return head;
}
```

```
Node* connectRandomPointers(Node* head){
    Node* temp = head;
    while(temp != NULL){
        Node* copy = temp->next;
        if(temp->random != NULL){
            copy->random = temp->random->next;
        }
        temp = temp->next->next;
    }
    return head;
}
```

```
Node* getDeepCopyList(Node* head){
    Node* temp = head;
    Node* dummy = new Node(-1);
    Node* res = dummy;
    while(temp != NULL){
        res->next = temp->next;
        res = res->next;
        temp->next = temp->next->next;
        temp = temp->next;
    }
    return dummy->next;
}
```

$T_c = O(3N)$

$S_c = O(N)$ → (as que. mentioned)

```
Node* copyRandomList(Node* head){
    insertCopyBetweenHead(head);
    connectRandomPointers(head);
    return getDeepCopyList(head);
}
```

Q) 3-SUM {3-target sum}

i != j != k, arr[i] + arr[j] + arr[k] == 0 } cond. must satisfy

Also, soln. set must not contain duplicate triplets.

A) 2nd Soln. {Better}: 3rd ele = 0 - (arr[i] + arr[j])

eg [-1 -1 -2 -1 -1 -1 0 0 0 2 2 2 2]

[, , , ,] (i) (j) → adding the 3rd ele. in set ← if already present { one of our answers }

insert in ans & insert nums[j] in set & move on

Code:

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

$T_c = N^2 \times \log M$ (M = size of hashset)

$S_c = O(N) + O(\text{no. of unique pairs}) \times 2$ ('ans' & external DS 'st')

3rd Soln. {optimal} → 1 pointer const, 2 pointers moving

```
vector<vector<int>> threeSum(vector<int>&nums){
    int n = nums.size();
    sort(nums.begin(), nums.end());
    vector<vector<int>> ans;
    for(int i=0; i<n; i++){
        if(i>0 and nums[i]==nums[i-1]){
            continue; //skipping duplicates
        }
        int j = i+1, k = n-1;
        while(j<k){
            int sum = nums[i] + nums[j] + nums[k];
            if(sum<0) j++;
            else if(sum>0) k--;
            else {
                vector<int> temp = {nums[i], nums[j], nums[k]};
                ans.push_back(temp);
                j++; k--;
                while(j<k and nums[j] == nums[j-1])
                { j++; }
                while(j<k and nums[k] == nums[k+1])
                { k--; }
            }
        }
    }
    return ans;
}
```

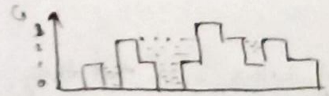
$T_c = N \log N + O(N^2)$

$S_c = O(\text{no. of unique trip.})$

Q) TRAPPING RAIN WATER

Given 'n' non-negative integers representing an elevation map where width of each bar is '1', compute how much water it can trap after raining.

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



ans 6

A) Soln 1 {Brute force} → at every index 'i' → calculate: (min(left(i), right(i)) - a[i])

left(i): left max before index 'i'; right(i): right max after index 'i'

$T_c = O(N^2)$

$S_c = O(1)$

Soln. 2 {Better} → calculate prefix sum & suffix sum

Here; pref (start from left): 0 1 1 2 2 2 2 3 3 3 3 3

suff (start from right): 3 3 3 3 3 3 3 3 2 2 2 1

• Now no need to calculate left(i), right(i) using for loops

$T_c = O(N) + O(N)$ (pref, suff) $+ O(N)$ (index traversal)

$S_c = O(2N)$

```
pref: for(i=1; i<n; i++){
          pref[i] = max(pref[i-1], height[i])
      }
suff: for(i=n-2; i>=0; i--){
          suff[i] = max(suff[i+1], height[i])
      }
```

Soln. 3: (Optimal) → 2 pointer approach:

$T_c = O(N)$
$S_c = O(1)$

```
int trap(vector<int>& height){
    int n = height.size();
    int left=0, right=n-1, res=0, maxleft=0, maxright=0;
    while(left <= right){
        if(height[left] <= height[right]){
            if(height[left] >= maxleft) maxleft = height[left];
            else res += (maxleft - height[left]);
            left++;
        }else{
            if(height[right] >= maxright) maxright = height[right];
            else res += (maxright - height[right]);
            right--;
        }
    }
    return res;
}
```

means no water can be stored at that block

since h[l] < h[r] & curr. bar height is less than maxleft, so we subtract & now we can say that it stores (subtract)'s value of water

Q) REMOVE OUTERMOST PARENTHESIS: ex: (()())(()) → op: ()()()
ex: (()()) → op: ()()

```
string removeOuterParenthesis(string s){
    int cnt = 0; string ans;
    for(int i=0; i<s.size(); i++){
        if(s[i] == '('){
            if(cnt > 0) ans += s[i]; //ignores outermost parenthesis
            cnt++;
        }
        else{
            cnt--;
            if(cnt > 0) ans += s[i];
        }
    }
    return ans;
}
```

$T_c = O(N)$
$S_c = O(N)$

Q) 2-SUM: array & target given. Return indices of 2 nos such that they add up to target. Using same element twice is not allowed.
(2-target-sum)

```
vector<int> twoSum(vector<int>& arr, int target){
    unordered_map<int,int> mp; int n = arr.size();
    for(int i=0; i<n; i++) mp[arr[i]] = i;
    for(int i=0; i<n; i++){
        int num = arr[i];
        int moreNeeded = target - num;
        if(mp.find(moreNeeded) != mp.end() and mp[moreNeeded] != i){
            return {i, mp[moreNeeded]};
        }
    }
    return {-1, -1};
}
```

$T_c = O(2N)$
$S_c = O(N)$

*If we use set, then it can return index of current ele. if current ele. = moreNeeded. We can store only nos & not indexes in set.

Q) 4-SUM: return array of all unique quadruplets: nums[a], nums[b], nums[c], nums[d]
(4-target-sum) such that: → 0 <= a,b,c,d < n → nums[a] + nums[b] + nums[c] + nums[d] = target
→ a,b,c,d are distinct.

ex: target = 8; arr = [1 1 1 2 2 2 3 3 3 4 4 4 5 5]

```
vector<vector<int>> fourSum(vector<int>& nums, int target){
    vector<vector<int>> ans;
    sort(nums.begin(), nums.end());
    int n = nums.size();
    for(int i=0; i<n; i++){
        if(i>0 and nums[i] == nums[i-1]) continue;
        for(int j=i+1; j<n; j++){
            if(j!=(i+1) and nums[j]==nums[j-1]) continue;
            int k = j+1, l = n-1;
            while(k < l){
                long long sum = nums[i];
                sum += nums[j];
                sum += nums[k];
                sum += nums[l];
                if(sum < target) k++;
                else if(sum > target) l--;
                else{
                    vector<int> temp = {nums[i], nums[j], nums[k], nums[l]};
                    ans.push_back(temp);
                    k++; l--;
                    while(k<l and nums[k]==nums[k-1]) k++;
                    while(k<l and nums[l]==nums[l+1]) l--;
                }
            }
        }
    }
    return ans;
}
```

Here 2 pointers const. 2 pointers moving

$T_c \approx O(n^3)$
$S_c = O(\text{ans.size()})$

to avoid runtime error i.e. overflow buffer error

Q) JOB-SEQUENCING PROBLEM:

Given set of 'N' jobs where each job has deadline & profit associated with it. Each job takes 1 unit of time to complete & only 1 job can be scheduled at a time.

Find no. of jobs done & max. profit.

ex: N=4, Jobs = {(1,4,20), (2,1,10), (3,1,40), (4,1,30)}
→ 2, 60 ↝ exp: (20+40) = 60

In soln.:

```
struct Job{
    int id;     → Job Id
    int dead;   → Deadline of Job
    int profit; → Profit if job is over before (or) on deadline
};
```

```
A) bool cmp(Job a, Job b){
       return a.profit > b.profit;
   }
   vector<int> JobScheduling(Job arr[], int n){
       sort(arr, arr+n, cmp); → sorting based on ↑ing order of Profits
       int maxi = arr[0].dead;
       for(int i=1; i<n; i++){
           maxi = max(maxi, arr[i].dead); → finding whose deadline is the maximum
       }
       int slot[maxi+1]; → creating an array of 'maxi' size, so that we can assign jobs performed on any given day
       for(int i=0; i<=maxi; i++) slot[i] = -1 → initializing with -1, as no jobs performed initially
       int countJobs=0, jobProfit=0;
       for(int i=0; i<n; i++){
           for(int j= arr[i].dead ; j>0 ; j--){
               if(slot[j]==(-1)){ → whichever slot is empty
                   slot[j] = i;
                   countJobs++;
                   jobProfit += arr[i].profit;
                   break;
               }
           }
       }
       return {countJobs, jobProfit};
   }
```

(Sorting) $T_c = O(N \log N) + O(N*M)$

iterate all N jobs & for each job, find deadline say 'M' deadlines in worst case

$S_c = O(M)$ → max. deadline

MINIMUM NO. OF COINS: Given ∞ supply of denominations: {1, 2, 5, 10, 20, 50, 100, 200, 500, 2000}, Find min. no. of coins needed to make change for Rs. 'N'.

```
A) vector<int> minPartition(int N){
       vector<int> deno = {1,2,5,10,20,50,100,200,500,2000};
       int sz = deno.size();
       vector<int> ans;
       for(int i=sz-1; i>=0; i--){
           while(N >= deno[i]){
               N -= deno[i];
               ans.push_back(deno[i]);
           }
       }
       return ans;
   }
```

- We are doing it here using 'greedy' as we can see sum of any 2 consecutive denomination is less than its next consecutive denomination

$T_c = O(sz)$

$S_c = O(ans.size())$

NUMBER OF COINS: Given value V, coins[] of size M, task is to make change for V cents given that you have ∞ supply of each coins. Find the min. no. of coins, to make the change.

ex: V=30, M=3, coins[] = {25, 10, 5} → op: 2 → ex: 25, 5

ex: V=11, M=4, coins[] = {9, 6, 5, 1}; op: 2 → op: 5, 6

A) Here you can see → 6+5 > 9 → so it's a hint, that we can't use greedy (as there is no uniformity b/w differences of 2 denominations)

∴ Applying DP to avoid TLE.

Soln. 1 (top-down): $T_c = O(N*T)$, $S_c = O(N*T)$

Rules:
1) f(index, target)
2) express all possibilities → take, notTake
3) Min. of all
4) Base Case

```
int f(vector<int>& coins, int index, int target, vector<vector<int>>& dp){
    if(index == 0){
        if(target % coins[index] == 0) return target/coins[index];
        else return 1e9;
    }
    if(dp[index][target] != -1) return dp[index][target];
    int notTake = 0 + f(coins, index-1, target, dp);
    int take = INT_MAX;
    if(coins[index] <= target) take = 1 + f(coins, index, target-coins[index], dp);
    return dp[index][target] = min(take, notTake);
}
int minCoins(vector<int>& coins, int n, int target){
    vector<vector<int>> dp(n, vector<int>(target+1, -1));
    int ans = f(coins, n-1, target, dp);
    if(ans >= 1e9) return -1;
    return ans;
}
```

(1e9 < INT_MAX)

Soln. 2: (Bottom-up) → 1) Base Case → 2) Check which var's are changing (index, target) → 3) Copy the recurrence

```
int minCoins(vector<int>& coins, int n, int target){
    vector<vector<int>> dp(n, vector<int>(target+1, 0));
    for(int t=0; t<=target; t++){ → target can be b/w 0 & target
        if(t % coins[0] == 0) dp[0][t] = t/coins[0];
        else dp[0][t] = 1e9;
    }
    for(int ind=1; ind<n; ind++){
        for(int t=0; t<=target; t++){
            int notTake = 0 + dp[ind-1][t];
            int take = INT_MAX;
            if(coins[ind] <= t){
                take = 1 + dp[ind][t-coins[ind]];
            }
            dp[ind][t] = min(take, notTake);
        }
    }
    int ans = dp[n-1][target];
    if(ans >= 1e9) return -1;
    return ans;
}
```

initially: idx → n-1, target → T] Bottom up: idx → 1, target → 0 to T

Q) COUNT NO. OF SUBARRAYS WITH GIVEN XOR 'K': Find total subarrays whose bitwise XOR of all elements equal to K.

A) Let xor = XR

1, 2, 3, 4, 5

Let xor = 'x' | Let xor = K

$\therefore x \wedge K = XR$

$\Rightarrow x = XR \wedge K$ } So we can find 'x' in map & get our answer.

Initially → add (0,1) to map ↳ map (frontXOR, its count)

Code:

```
int solve(vector<int>& a, int b){
    int xr=0;
    unordered_map<int,int> mp;
    mp[xr]++;
    int count=0;
    for(int i=0; i<a.size(); i++){
        xr = xr ^ a[i];
        int x = xr ^ b;
        count += mp[x];
        mp[xr]++;
    }
    return count;
}
```

insertion → O(1) → best case, O(N) → worst case

dry run: [4, 2, 2, 6, 4]

XR = 4, 6, 4, 2, 6

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

cnt = 1, 3, 4

$T_c = O(N)$

$S_c = O(N)$

Q) LONGEST SUBSTRING WITHOUT REPEATING CHARACTERS: ex: "abcabcbb" → op: 3 → "abc"; ex: "bbbbb" → op: 1; ex: "pwwkew" → op: 3

ex: abba → op: 2

A)
```
int lenLongestSubstring(string s){
    if(s.size()==0) return 0;
    unordered_map<int,int> mp; → to store last index of each character
    int ans=0; → to store length of longest substring
    int left=0; → left index of sliding window
    for(int right=0; right<s.size(); right++){
        if(mp.find(s[right]) != mp.end() and mp[s[right]] >= left){
            left = mp[s[right]] + 1; → move left ptr to right of last occurence
        }
        mp[s[right]] = right; → update last occurence of character
        ans = max(ans, right-left+1);
    }
    return ans;
}
```

If char is found in map & its index is within the current window

$T_c = O(N)$

$S_c = O(N)$

Q) MINIMUM PLATFORMS: We are given 2 arrays = arrival & 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. At any given instance of time, same platform can't be used for both departure of train & arrival of train. In such cases, we need diff. platforms.

ex: arr[] = {0900, 0940, 0950, 1100, 1500, 1800}, dept[] = {0910, 1200, 1120, 1130, 1900, 2000}

Time = 24hr format

A) Here we will sort both arrays → to ensure correct determination of the no. of platforms needed at any given time (since the arrival & departure aren't associated with any particular train, so we can sort)

Code:
```
int findPlatform(int arr[], int dep[], int n){
    sort(arr, arr+n); sort(dep, dep+n);
    int maxPlat = 1, plat = 1;  int i = 1, j = 0;
    while(i<n and j<n){
        if(arr[i] <= dep[j]){
            plat++; i++;
        }
        else{
            plat--; j++;
        }
        maxPlat = max(maxPlat, plat);
    }
    return maxPlat;
}
```

$T_c = O(n \log n) + O(n)$

$S_c = O(1)$

Brute force: (2 for loops) ↳
```
ans=1;
for(i=0 to n-1){
    count=1;
    for(j=i+1 to n-1){
        if((arr[i]>=arr[j] and arr[i]<=dep[j]) or
           (arr[j]>=arr[i] and arr[j]<=dep[i])){
            count++;
    }}
    ans = max(ans, count);
}
return ans;
```

$T_c = O(n^2)$

$S_c = O(1)$

Q) STRING COMPRESSION III

Use this algo. to reduce given string: → Empty string 'comp'. While 'word' is not empty:
(→ remove max length prefix of 'word' made by a single char. 'c' repeating atmost 9 times.
→ append length of prefix followed by 'c' to comp

Return 'comp'.

ex: "abcde" → "1a1b1c1d1e"

ex: "aaaaaaaaaaaaaabb" → "9a5a2b"

A)
```
string compressedString(string word){
    string comp;
    int idx=0, n=word.size();
    while(idx<n){
        char c = word[idx];
        int cnt=0;
        while(idx<n and cnt<9 and word[idx]==c){
            idx++;
        }  cnt++;
        comp += to_string(cnt) + c;
    }
    return comp;
}
```

$T_c = O(N)$ (string size)

$S_c = O(N)$ (needed in ques.)

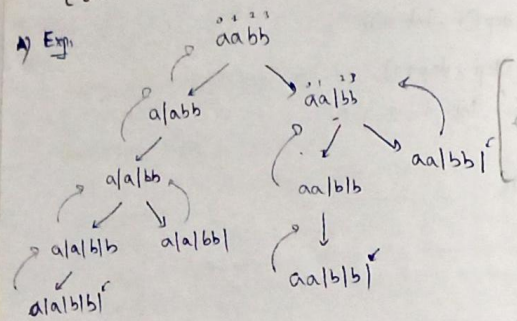
Q) PALINDROME PARTITIONING

: Given string 's', partition 's' such that every substring of partition is palindrome. Return all possible palindrome partitioning.

Ex: s = "aabb"

O/P: [["a","a","b","b"], ["a","a","bb"], ["aa","b","b"], ["aa","bb"]]

A) Exp:



$T_c = O(2^n \cdot k \cdot (n/2))$

(generating every substring) × (for checking palindrome in another doubly chr.) × (check if substring is palindrome or not)

$S_c = O(k * x)$

(avg. len. of palin.) × ('x' such palindromes)

Code:

```
bool isPalindrome(string s, int start, int end){
    while(start <= end){
        if(s[start++] != s[end--]) return false;
    }
    return true;
}
void func(int index, string s, vector<string>& path, vector<vector<string>>& res){
    if(index == s.size()){
        res.push_back(path);
        return;
    }
    for(int i=index; i<s.size(); i++){
        if(isPalindrome(s, index, i)){
            path.push_back(s.substr(index, i-index+1));
            func(i+1, s, path, res);
            path.pop_back();
        }
    }
}
vector<vector<string>> partition(string s){
    vector<vector<string>> res;
    vector<string> path;
    func(0, s, path, res);
    return res;
}
```

Q) PERMUTATION SEQUENCE

Set [1,2,3,... n] contains a total of n! unique permutations. Given n' & 'k', return the k^th permutation sequence.

Ex: n=3, k=3 → 123, 132, **213** ✓, 231, 312, 321

A) Better Soln:

```
string getPermutation(int n, int k){
    string res = "";
    int dus[n];
    for(int i=1; i<=n; i++) dus[i-1] = i;
    for(int i=1; i<k; i++){
        next_permutation(dus, dus+n);
    }
    for(int i=0; i<n; i++) res += to_string(dus[i]);
    return res;
}
```

$T_c = O(n * k)$

$S_c = O(res.size())$

Optimal Soln:

```
string getPermutation(int n, int k){
    int fact = 1;
    vector<int> nums;
    for(int i=1; i<n; i++){
        fact = fact * i;
        nums.push_back(i);
    }
    nums.push_back(n);
    string ans = "";
    k = k-1;
    while(true){
        ans += to_string(nums[k/fact]);
        nums.erase(nums.begin() + k/fact);
        if(nums.size() == 0) break;
        k = k % fact;
        fact = fact / nums.size();
    }
    return ans;
}
```

$T_c = O(n * n)$ (while loop) (erase ele.)

Q) N-QUEENS

A) Soln. 1: Refer notes → $T_c = N! \times N$, $S_c = O(N^2)$

Soln. 2: → $T_c = N! \times N$, $S_c = O(N)$

Code:

```
bool canPlaceQueen(int row, int col, vector<string>& board, vector<int>& leftRow,
    vector<int>& lowerDiagonal, vector<int>& upperDiagonal){
    if(leftRow[row]==1 or lowerDiagonal[row+col]==1 or upperDiagonal[board.size()-1+col-row]
        ==1){ return false; }
    return true;
}
void nqueen(int col, int n, vector<string>& board, vector<vector<string>>& result, vector<int>&
    leftRow, vector<int>& lowerDiagonal, vector<int>& upperDiagonal){
    if(col == n){
        result.push_back(board);
        return;
    }
    for(int row=0; row<n; row++){
        if(canPlaceQueen(row, col, board, leftRow, lowerDiagonal, upperDiagonal)){
            board[row][col] = 'Q';
            leftRow[row] = 1;
            lowerDiagonal[row+col] = 1;
            upperDiagonal[n-1+col-row] = 1;
            nqueen(col+1, n, board, result, leftRow, lowerDiagonal, upperDiagonal);
            board[row][col] = '.';
            leftRow[row] = 0;
            lowerDiagonal[row+col] = 0;
            upperDiagonal[n-1+col-row] = 0;
        }
    }
}
```

```
vector<vector<string>> solveNQueens(int n){
    vector<vector<string>> result;
    vector<string> board(n, string(n,'.'));
    vector<int> leftRow(n,0), lowerDiagonal(2*n-1,0), upperDiagonal(2*n-1,0);
    nqueen(0, n, board, result, leftRow, lowerDiagonal, upperDiagonal);
    return result;
}
```

A) N

Q) WORD-BREAK-2 : Given non-empty sequence s & a dictionary dict[] containing a list of non-empty words, print all possible ways to break the sentence in individual dictionary words.

ex: s = "catsanddog"
dict[] = {"cat","cats","and","sand","dog"} } wordSet → "cats and dog" / "cat sand dog"

A)
```
void wordBreak(string &s, unordered_set<string>& wordSet, vector<string>& ans, string temp, int start){
    if(start == s.size()){
        temp.pop_back();  → remove trailing space
        ans.push_back(temp);
        return;
    }
    for(int i=start; i<s.size(); i++){
        string word = s.substr(start, i-start+1);
        if(wordSet.find(word) != wordSet.end()){
            wordBreak(s, wordSet, ans, temp+word+" ", i+1);
        }
    }
}
vector<string> wordBreak2(string &s, vector<string>& dictionary){
    unordered_set<string> wordSet(wordDict.begin(), wordDict.end());   (easy in searching & adding)
    wordBreak(s, wordSet, ans, "", 0);
    return ans;
}
```

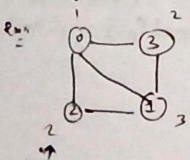
$T_c = O(n \cdot m \cdot 2^n)$ — worst case of 'substr' func. · (total substrings) · ('find' func. time) · m = size of dict[]

$S_c = O(n \cdot 2^n)$ — (recursion depth) · (storing 'ans' result)

Q) M-COLORING PROBLEM:

Given an undirected graph & an int. M. Determine if graph can be coloured with atmost M colors such that no 2 adjacent vertices of graph are coloured with same color. Here coloring of a graph means the assign. of colors to all vertices. Print 1 if possible, else 0.

ex: If N=4, M=3, E=5, Edges = {(1,2),(2,3),(3,4),(4,1),(1,3)}

ex: 

→ ans: yes = 1

```
bool isSafe(int node, int color[], bool graph[101][101], int n, int col){
    for(int k=0; k<n; k++){   → traversing all adjacent nodes
        if(k!=node and graph[k][node]==1 and color[k]==col) return false;   (there is an edge, so it's adjacent)
    }
    return true;
}
bool solve(int node, int color[], int m, int n, bool graph[101][101]){
    if(node==n) return true;
    for(int i=1; i<=m; i++){   → trying out every color
        if(isSafe(node, color, graph, n, i)){   → tells if it's possible to color that node → checks adjacent nodes
            color[node]=i;
            if(solve(node+1, color, m, n, graph)) return true;
            color[node]=0;
        }
    }
    return false;
}
bool graphColoring(bool graph[101][101], int m, int n){   (no. of color) (no. of nodes)
    int color[n]={0};   → none of the nodes are colored initially
    if(solve(0, color, m, n, graph)) return true;
    return false;
}
```

graph[i][j] = 1 (means edge there btw i & j)

$T_c = (N^m) \cdot n$ — 'm' colors, 'n' nodes → trying all colors of n nodes → backtracking

$S_c = O(N)$ → depth of recursion tree

Q) K-th ELEMENT OF 2 SORTED ARRAYS : Find the ele. that would be at kth position of the final sorted array.

ex: a1[] = {2,3,6,7,9}, a2[] = {1,4,8,10}, k=5 } ex: 6 → ex: 1,2,3,4,6,7,8,9,10

A)
```
int kthElement(int arr1[], int arr2[], int n, int m, int k){
    if(n>m) return kthElement(arr2, arr1, m, n, k);
    int lo = max(0, k-m), hi = min(k, n);
    while(lo<hi){
        int cut1 = (lo+hi)>>1;
        int cut2 = k-cut1;
        int l1 = (cut1==0)?INT_MIN:arr1[cut1-1];
        int l2 = (cut2==0)?INT_MIN:arr2[cut2-1];
        int r1 = (cut1==n)?INT_MAX:arr1[cut1];
        int r2 = (cut2==m)?INT_MAX:arr2[cut2];
        if(l1>r2) hi=cut1-1;
        else if(l2>r1) lo=cut1+1;
        else return max(l1,l2);
    }
    return 1;
}
```

We are applying BS on shorter array

if k>m, i.e. arr1.size() then we need min. k-m elements in our answer
ex: n=5, n=10, k=7

need at least 2 ele. & then we can start with lower bound

if k>n, i.e. arr2.size(), then we need min. of both to keep upper bound of BS

$T_c = O(\log(\min(n,m)))$, $S_c = O(1)$

Q) AGGRESSIVE COWS: Given arr → size 'n' = denotes position of stalls. K = no. of aggressive cows. Assign stalls to 'K' cows such that the min. distance btw any 2 of them is max. · Find maximum possible minimum distance.

ex: N = 6, K = 4, arr[] = {0, 3, 4, 7, 10, 9} → op: 3 → ex: cows at: 0, 3, 7, 10

```
A) bool canWePlace(vector<int>& stalls, int n, int mid, int k){
    int currCows = 1, last = stalls[0];
    for(int i=1; i<n; i++){
        if(stalls[i] - last >= mid){
            currCows++;
            last = stalls[i];
        }
    }
    if(currCows < k) return false;
    return true;
}
int solve(int n, int k, vector<int>& stalls){
    sort(stalls.begin(), stalls.end());
    int lo=0, hi = stalls[n-1] - stalls[0];
    int ans = 0;
    while(lo <= hi){
        int mid = (lo+hi)/2;
        if(canWePlace(stalls, n, mid, k) == true){
            ans = mid;
            lo = mid+1;
        }
        else hi = mid-1;
    }
    return ans;
}
```

$T_c = O(n \log n) + O(n \times \log_2(h))$ (sorting)

→ biggest - lowest

$S_c = O(1)$

Q) LEXOGRAPHICALLY MINIMUM STRING AFTER REMOVING STARS

Given string 's'. It may contain any no. of '*' characters. You have to remove all '*' characters.

While there is a '*', do:

→ delete the leftmost '*' & the smallest non-'*' character to its left. If there are several smallest characters, delete any of them.

Return lexographically smallest resulting string after removing all '*' characters.

ex: 1) 'aaba*' → op: aab

2) 'abc' → op: abc

3) 'de*' → op: e

A) We will use 2 data str.'s here;

→ Stack ('st'): used to maintain order of characters as they appear in the string

→ MultiSet ('charset'): used to efficiently find & remove the smallest lexographical character.

```
string clearStars(string s){
    stack<char> st;
    multiset<char> charSet;
    for(char c : s){
        if(c == '*'){
            if(!charSet.empty()){
                char smallest = *charSet.begin();
                charSet.erase(charSet.begin());
                stack<char> tempStack;
                while(!st.empty() and st.top() != smallest){
                    tempStack.push(st.top());
                    st.pop();
                }
                if(!st.empty()) st.pop();
                while(!tempStack.empty()){
                    st.push(tempStack.top());
                    tempStack.pop();
                }
            }
        }
        else{
            st.push(c);
            charSet.insert(c);
        }
    }
    string result;
    while(!st.empty()){
        result += st.top();
        st.pop();
    }
    reverse(result.begin(), result.end());
    return result;
}
```

(returns an iterator, so using *)

→ remove the smallest lexographical character

$T_c = O(\log N)$

remove smallest char. from stack ↓

$T_c = O(N)$

Push non-'*' character onto the stack & add to multiset

$T_c = O(\log N)$

collect characters from stack to form result

→ reverse the result as chars are collected in reverse order

$T_c = O(N) + O(N \log n) = O(n \log n)$

(stack) (multiset)

$S_c = O(n) + O(n) = O(n)$

Dry run:

a) "de*" (initially empty)

1) st: 'd'
charset: {'d'}

2) st: ['d', 'e']
charset: {'d', 'e'}

3) → remove 'd' from charset
→ remove leftmost 'd' from stack & store ele.'s in temp. stack.
→ Again re-add ele.'s to stack

4) st: ['e']
charset: {'e'}

b) "aaba*" : 1), 2), 3) st ['a', 'a', 'b'] (initially empty) charset {'a', 'a', 'b'}

4) st: ['a', 'a', 'b', 'a']
charset: {'a', 'a', 'a', 'b'}

5) → remove 'a' from charset as it's smallest one
→ remove leftmost 'a' from stack
→ temp. stack to remove chars. until 'a' is found
→ Pop 'a' from st' → st: ['a', 'a', 'b']
→ re-add ele.'s

6) st: ['a', 'a', 'b']
charset: {'a', 'a', 'b'}

Q) **MAXIMUM SUM COMBINATIONS** : Given 2 equally sized 1-D arrays A, B containing N integers each. A sum combination is made by adding 1 element from array A & another element of array B.
Return max value 'C' valid sum combinations from all the possible sum combinations.

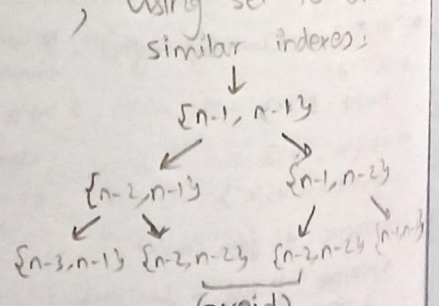
ex: A = [1,4,2,3] , B = [2,5,1,6] , C = 4 → o/p: [10,9,9,8]

A)
```
vector<int> solve (vector<int> &a, vector<int> &b, int c){
    int n = a.size();
    sort(a.begin(), a.end()); sort(b.begin(), b.end());
    priority_queue<pair<int, pair<int,int>>> pq; → storing sum & indexes of both arrays
    set<pair<int,int>> s;
    pq.push({a.back()+b.back(), {n-1, n-1}});
    s.insert({n-1, n-1}); → maintaining indexes
    vector<int> ans;
    while(c--){
        auto p = pq.top();
        int sum = p.first;
        int i = p.second.first, j = p.second.second;
        ans.push_back(sum);
        pq.pop();
        if(s.find({i-1, j}) == s.end()){
            pq.push({a[i-1]+b[j], {i-1, j}});
            s.insert({i-1, j});
        }
        if(s.find({i, j-1}) == s.end()){
            pq.push({a[i]+b[j-1], {i, j-1}});
            s.insert({i, j-1});
        }
    }
    return ans;
}
```

$T_c = K\log n + n\log n$, $S_c = O(n)$

(while loop) (insertion in set & pq) (for sorting) (set size)

, Using set to avoid similar indexes:



Q) **SEGMENTED SIEVE** : Generate prime no.'s in a small range but btw high values

ex: Prime no.'s btw 10^7 & 10^9.

A)
```
vector<bool> sieve(int n){
    vector<bool> isPrime(n+1, true);
    isPrime[0] = isPrime[1] = false;
    for(int i=2; i*i<=n; i++){
        if(isPrime[i]){
            for(int j=i*i; j<=n; j+=i){
                isPrime[j] = false;
            }
        }
    }
    return isPrime;
}

vector<bool> getPrimesInRange(long long l, long long r){
    vector<bool> prime = sieve(sqrt(r));
    vector<bool> isPrime(r-l+1, true);
    for(long long i=2; i*i<=r; i++){
        if(!prime[i]) continue;
        for(long long j=max(i*i, (l+i-1)/i*i); j<=r; j+=i){ // multiple of 'i' in range btw (l,r)
            isPrime[j-l] = false;
        }
    }
    if(l==1) isPrime[0] = false;
    return isPrime;
}
```

```
int main(){
    long long l,r; cin>>l>>r;
    vector<bool> ans = getPrimesInRange(l,r);
    for(int i=0; i<ans.size(); i++){
        if(ans[i] == true){
            cout<<l+i<<" ";
        }
    }
    return 0;
}
```

$T_c = O((r-l-1)\log(\log(r)) + \lceil r\log(\log(\lceil r\rceil))$, $S_c = O(r-l-1)$

Q) **TOP K FREQUENT ELEMENTS** : Given array, return 'k' most frequent ele.'s

ex: [1,1,1,2,2,3], k=2 → o/p: [1,2]

A)
```
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>>> pq;
    for(auto i:nums) mp[i]++;
    for(auto i:mp){
        pq.push({i.second, i.first}); → frequency & element. Sorted based on freq.
        if(pq.size() > k) pq.pop();
    }
    vector<int> ans;
    while(k--){
        ans.push_back(pq.top().second);
        pq.pop();
    }
    return ans;
}
```

$T_c = n.\log k$ → insertion time in minheap

$S_c = O(k)$

// If maxheap, then we can get 'k' topmost ele.'s but there $T_c = O(n\log n)$ & $S_c = O(n)$

∴ Minheap is optimised approach.

Q) NEXT GREATER ELEMENT USING STACK

Given circular array A, return next greater element (nge) in A. NGE of the 1st element > than the current ele. when traversed in a clockwise manner. If doesn't exist, return -1.

ex: A[] = {5,7,1,2,6,0} → if circular array: {7,-1,2,6,7,5}
→ if non-circular array: {7,-1,2,6,-1,-1}

A) Circular:

```
vector<int> nGE(vector<int>& nums){
    int n=nums.size();
    vector<int> nge(n,-1);
    stack<int> st;
    for(int i=2*n-1; i>=0; i--){
        while(!st.empty() and st.top()<=nums[i%n]){
            st.pop();
        }
        if(!st.empty()){
            nge[i%n]=st.top();
        }
        st.push(nums[i%n]);
    }
    return nge;
}
```

$T_c = O(N)$, $S_c = O(N)$

- If needed for non-circular array
  - change → `for(int i=n-1; i>=0; i--)`
    - i%n ⇒ 'i'

Q) SORT STACK USING RECURSION ; ex: 5 -2 9 -7 3 → 9 5 3 -2 -7

A)
```
void sortStack(stack<int>& st){
    if(st.size()==1) return;
    int a=st.top();
    st.pop();
    sortStack(st);
    stack<int> st2;
    while(!st.empty() and a<st.top()){
        st2.push(st.top());
        st.pop();
    }
    st.push(a);
    while(!st2.empty()){
        st.push(st2.top());
        st2.pop();
    }
}
```

$T_c = O(n)$

$S_c = O(N)$

Q) COUNT OF SMALLER NUMBERS AFTER SELF: Given 'nums[]', return 'counts[]' where counts[i] = no. of smaller ele.s to right of nums[i]. [Similar prob = 'count Inversion']

ex: [5,2,6,1] → [2,1,1,0]

A)
```
void merge(vector<int>& ans, vector<pair<int,int>>& numsPair, int left, int mid, int right){
    vector<pair<int,int>> temp(right-left+1);
    int i=left, j=mid+1, k=0;
    while(i<=mid and j<=right){
        if(numsPair[i].first <= numsPair[j].first){ temp[k++]=numsPair[j++]; }
        else{
            ans[numsPair[i].second] += right-j+1;
            temp[k++]=numsPair[i++];
        }
    }
    while(i<=mid) temp[k++]=numsPair[i++];
    while(j<=right) temp[k++]=numsPair[j++];
    for(int i=left; i<=right; i++) numsPair[i]=temp[i-left];
}
void mergeSort(vector<int>& ans, vector<pair<int,int>>& numsPair, int left, int right){
    if(left<right){
        int mid=(left+right)/2;
        mergeSort(ans, numsPair, left, mid);
        mergeSort(ans, numsPair, mid+1, right);
        merge(ans, numsPair, left, mid, right);
    }
}
vector<int> countSmaller(vector<int>& nums){
    int n=nums.size();
    vector<pair<int,int>> numsPair;
    for(int i=0; i<n; i++) numsPair.push_back({nums[i], i});
    vector<int> ans(n,0);
    mergeSort(ans, numsPair, 0, n-1);
    return ans;
}
```

$T_c = O(n \log n)$, $S_c = O(n)$

Q) NEXT SMALLER ELEMENT: ex: [4,5,2,10,8] → o/p: [-1,4,-1,2,2]

A)
```
vector<int> smaller(vector<int>& v){
    vector<int> ans(v.size(),-1);
    stack<int> st;
    for(int i=0; i<v.size(); i++){
        while(!st.empty() and st.top()>=v[i]) st.pop();
        if(st.empty()) ans[i]=st.top();
        st.push(v[i]);
    }
    return ans;
}
```

$T_c = O(n)$, $S_c = O(n)$

Q) LRU CACHE: {Least Recently Used}

- get(key) → value (or -1)
- put(key, value) → remove the LRU (any) & then insert

ex: size = 2 ; put(1,1) → {1,1}
put(2,2) → {1,1},{2,2}
get(1) → 1 (as '1' was accessed earlier, so LRU = {2,2})
put(3,3) → {1,1},{3,3}
get(2) → -1
put(4,4) → {3,3},{4,4}
get(1) → -1
get(3) → 3
get(4) → 4

{refer video for dry run of DS}

A) struct Node {
int key; int val;
node* next; node* prev;
node(int _key, int _val){
key = _key;
val = _val;
}
};
node* head = new node(-1,-1);
node* tail = new node(-1,-1);
int cap;
unordered_map<int, node*> m;

```
LRUCache(int capacity){
    cap = capacity;
    head->next = tail;
    tail->prev = head;
}

void addnode(node* newnode){
    node* temp = head->next;
    newnode->prev = head;
    newnode->next = temp;
    head->next = newnode;
    temp->prev = newnode;
}

void deletenode(node* delnode){
    node* delprev = delnode->prev;
    node* delnext = delnode->next;
    delprev->next = delnext;
    delnext->prev = delprev;
}

int get(int key_){
    if(m.find(key_) != m.end()){
        node* resnode = m[key_];
        int res = resnode->val;
        m.erase(key_);
        deletenode(resnode);
        addnode(resnode);
        m[key_] = head->next;
        return res;
    }
    return -1;
}

void put(int key_, int value){
    if(m.find(key_) != m.end()){
        node* existingnode = m[key_];
        m.erase(key_);
        deletenode(existingnode);
    }
    if(m.size() == cap){
        m.erase(tail->prev->key);
        deletenode(tail->prev);
    }
    addnode(new node(key_, value));
    m[key_] = head->next;
}
```

Tc = O(n)
Sc = O(1)

DS = Hashmap, doubly ll

Q) LFU CACHE: {least Frequently Used}

- get(key) → gets the value of key if exists, else -1
- put(key, value) → update the value of key if present (or) insert the key if not present

When the cache is full, it removes the LFU one & if tie, then the LRU one

ex: size = 2 ; put(1,10) ⇒ freq 1 → (1,10)
put(2,20) ⇒ 1 → (1,10),(2,20)
get(1) ⇒ 1 → (2,20)
2 → (1,10)
put(3,30) ⇒ 1 → (3,30) (as '2' was LFU)
2 → (1,10)
get(2) ⇒ -1
get(3) ⇒ 30 → 1 → 2 → (1,10)(3,30)
put(4,4) ⇒ 1 → (4,4)
2 → (3,30) (tie btw '1' & '3', so removed LRU i.e. → '1')
get(1) → -1
get(3) → 30 → 1 → (4,4)
2 → 3 → (3,30)
get(4) → 4 → 1 → 2 → (4,4)
3 → (3,30)

Tc = O(1)
Sc = O(n)

A) struct Node{
int key, value, cnt;
Node * next;
Node * prev;
Node(int _key, int _value){
key = _key;
value = _value;
cnt = 1;
}
};
struct List{
int size;
Node *head;
Node * tail;
List(){
head = new Node(0,0);
tail = new Node(0,0);
head->next = tail;
tail->prev = head;
size = 0;
}
void addFront(Node* node){
Node* temp = head->next;
node->next = temp;
node->prev = head;
head->next = node;
temp->prev = node;
size++;
}
void removeNode(Node* delnode){
Node* delprev = delnode->prev;
Node* delnext = delnode->next;
delprev->next = delnext;
delnext->prev = delprev;
size--;
};

```
Map<int, Node*> keyNode;
Map<int, List*> freqListMap;
int maxSizeCache, minFreq, curSize;
LFUCache(int capacity){
    maxSizeCache = capacity;
    minFreq = 0;
    curSize = 0;
}
void updateFreqListMap(Node* node){
    keyNode.erase(node->key);
    freqListMap[node->cnt]->removeNode(node);
    if(node->cnt == minFreq && freqListMap[node->cnt]->size == 0){
        minFreq++;
    }
    List* nextHigherFreqList = new List();
    if(freqListMap.find(node->cnt+1) != freqListMap.end()){
        nextHigherFreqList = freqListMap[node->cnt+1];
    }
    node->cnt += 1;
    nextHigherFreqList->addFront(node);
    freqListMap[node->cnt] = nextHigherFreqList;
    keyNode[node->key] = node;
}
```

```
int get(int key){
  if(keyNode.find(key) != keyNode.end()){
    Node* node = keyNode[key];
    int val = node->value;
    updateFreqListMap(node);
    return val;
  }
  return -1;
}
```

```
void put(int key, int value){
  if(maxSizeCache == 0) return;
  if(keyNode.find(key) != keyNode.end()){
    Node* node = keyNode[key];
    node->value = value;
    updateFreqListMap(node);
  }
  else{
    if(currSize == maxSizeCache){
      List* list = freqListMap[minFreq];
      keyNode.erase(list->tail->prev->key);
      freqListMap[minFreq]->removeNode(list->tail->prev);
      curSize--;
    }
    curSize++;
    minFreq = 1;   // new value has to be added which is not there previously
    List* listFreq = new List();
    if(freqListMap.find(minFreq) != freqListMap.end()){
      listFreq = freqListMap[minFreq];
    }
    Node* node = new Node(key, value);
    listFreq->addFront(node);
    keyNode[key] = node;
    freqListMap[minFreq] = listFreq;
  }
}
```

Q) REVERSE WORDS IN A STRING

ex: " hello world " → o/p: "world hello"

```
A) string reverseWords(string s){
    string ans;
    for(int i=s.size()-1; i>=0; i--){
      while(i>=0 and s[i]==' ') i--;
      string word;
      while(i>=0 and s[i]!=' '){
        word += s[i];
        i--;
      }
      if(word.size()>=1){
        if(!ans.empty()) ans += ' ';
        reverse(word.begin(), word.end());
        ans += word;
      }
    }
    return ans;
}
```

$T_c = O(n)$, $S_c = O(n)$

Q) LONGEST PALINDROMIC SUBSTRING

ex: "cbbd" → "bb"
"babad" → "bab" (or "aba")

```
A) string longestPalindrome(string s){
    if(s.size()==1) return s;
    int max_len=1, n=s.size(), start=0;
    for(int i=0; i<n; i++){
      int l=i, r=i;
      while(l>=0 and r<n and s[l]==s[r]){
        l--; r++;
      }
      int len = r-l-1;
      if(len>max_len){
        max_len = len;
        start = l+1;
      }
    }
    return s.substr(start, max_len);
}
```

```
for(int i=0; i<n-1; i++){
  int l=i, r=i+1;
  while(l>=0 and r<n and s[l]==s[r]){
    l--; r++;
  }
  int len = r-l-1;
  if(len > max_len){
    max_len = len;
    start = l+1;
  }
}
```

KMP Algo. → $T_c = O(n)$

$T_c = O(n^2)$
$S_c = O(1)$

Q) STRING TO INTEGER (ATOI)

ex: "42" → 42 , " -042" → -42
"1337c0d3" → 1337 , "0-1" → 0 , "words987" → 0

Convert string to 32-bit signed integer

```
A) int myAtoi(string s){
    int i=0;
    while(i<s.size() and s[i]==' ') i++;
    if(i>=s.size()) return 0;
    int sign = 1;
    if(s[i]=='+' or s[i]=='-'){
      if(s[i]=='-') sign = -1;
      i++;
    }
    if(i>=s.size()) return 0;   // whole 's' consists of '+' or '-'
    long long num = 0;
    for(; i<s.size() and isdigit(s[i]); i++){
      int digit = s[i]-'0';   // Checking overflow/underflow condition
      if(num > INT_MAX/10 or (num==INT_MAX/10 and digit > INT_MAX%10)){
        return (sign==1) ? INT_MAX : INT_MIN;
      }
      else if(num < INT_MIN/10 or (num==INT_MIN/10 and digit > -(INT_MIN%10))){
        return (sign==1) ? INT_MAX : INT_MIN;
      }
      num = num*10 + digit;
    }
    num = num*sign;
    return num;
}
```

$T_c = O(n)$
$S_c = O(1)$

Q) REPEATED STRING MATCH

Return min. of times you should repeat string 'a' so that string 'b' is substring of it.

ex: a="abcd", b="cdabcdab"
o/p: 3 → a: abcdabcdabcd

4 cases: b = prefix + n*a + suffix ⇒ cd(abcd)ab → n+2
b = prefix + n*a ⇒ cd(abcd) → n+1
b = n*a + suffix ⇒ (abcd)ab → n+1
b = n*a ⇒ (abcd) → n

A) Brute force:

```
int repeatedStringMatch(string a, string b){
    int len_a = a.size(), len_b = b.size();
    int n = len_b/len_a;
    int cnt = n; string na = "";
    while(cnt--) na = na+a;
    if(na.find(b) != string::npos) return n;
    na = na+a;
    if(na.find(b) != string::npos) return n+1;
    na = na+a;
    if(na.find(b) != string::npos) return n+2;
    return -1;
}
```

$T_c = O(n+m)$ (find)
(size of str 'a') (size of str 'b')
$S_c = O(1)$

For optimal → refer KMP Method in front of book
↳ refer next page

Optimal: { Rabin Karp Method }

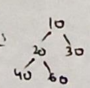
```
int repeatedStringMatch(string a, string b){
    if(a==b) return 1;
    int count = 1;
    string source = a;
    while(source.size() < b.size()){
        count++;
        source+=A;
    }
    if(source==b) return count;
    if(Rabin_Karp(source,b)!=-1) return count;
    if(Rabin_Karp(source+a, b)!=-1) return count+1;
    return -1;
} int BASE = 1000000;
```

$T_c = O(N+M)$

$S_c = O(N+M)$

```
int Rabin_Karp(string source, string target){
    if(source=="" or target=="") return -1;
    int m= target.size(), power=1;
    for(int i=0; i<m; i++) power=(power*31)%BASE;
    int targetCode=0;
    for(int i=0; i<m; i++) targetCode= (targetCode*31 + target[i])%BASE;
    int hashCode=0;
    for(int i=0; i<source.size(); i++){
        hashCode=((hashCode*31)+ source[i])%BASE;
        if(i<m-1) continue;
        if(i>=m) hashCode = (hashCode - source[i-m]*power)%BASE;
        if(hashCode<0) hashCode += BASE;
        if(hashCode == targetCode){
            if(source.substr(i-m+1,m)==target) return i-m+1;
        }
    }
    return -1;
}
```

Q) ROOT TO LEAF PATHS : ex:



→ op: 10 20 40
10 20 60
10 30

```
A) vector<vector<int>> ans;
void path(Node*path, vector<int>&temp){
    temp.push_back(root->data);
    if(root->left==NULL and root->right==NULL){
        ans.push_back(temp);
        temp.pop_back();
        return;
    }
    if(root->left!=NULL) path(root->left, temp);
    if(root->right!=NULL) path(root->right, temp);
    temp.pop_back();
}
```

```
vector<vector<int>> Paths(Node*root){
    vector<int> temp;
    if(root==NULL) return ans;
    path(root, temp);
    return ans;
}
```

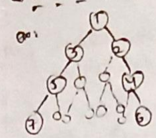
$T_c = O(n)$

$S_c = O(n)$

Q) MAXIMUM_WIDTH_OF_BINARY_TREE

Answer in range of 32-bit integer

ex:



op: 8

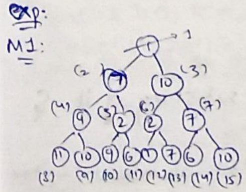
```
A) int widthOfBinaryTree(Node* root){
    if(root==NULL) return 0;
    int ans=0;
    queue<pair<Node*, int>> q;   // Node* → for level order traversal; int → position in the level
    q.push({root,0});
    while(!q.empty()){
        int size= q.size();
        int mn= q.front().second;
        int first, last;   → storing first & last position of nodes in curr. level
        for(int i=0; i<size; i++){   → Process each node at current level
            int cur_id = q.front().second - mn;   → calculate curr. pos. relative to min. position in the level
            Node* nod= q.front().first;
            q.pop();
            if(i==0) first = cur_id   → first node at current level
            if(i==size-1) last= cur_id;   → last node at current level
            if(node->left!=NULL) q.push({nod->left, (long long)cur_id*2 + 1});
            if(node->right!=NULL) q.push({nod->right, (long long)cur_id*2 +2});
        }
        ans=max(ans, last-first+1);
    }
    return ans;
}
```

$T_c = O(N)$

$S_c = O(N)$

exp:

M1:

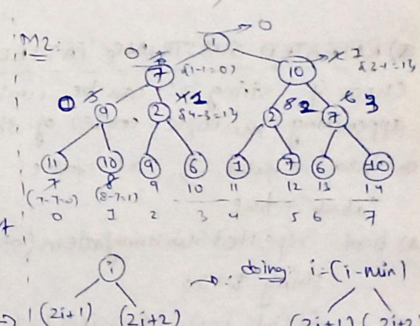


Level order traversal

Width = 15 - 8 + 1 = 8

Just keeps 'adding 1' & subtract start & end level
↓
Issue: Might cause overflow issue ⟹

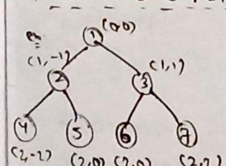
M2:



→ doing i-(i-min)

Q) VERTICAL ORDER TRAVERSAL OF A BINARY TREE

ex:

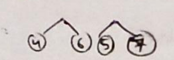


root = [1,2,3,4,5,6,7]

op: [[4],[2],[1,5,6],[3],[7]]

Note: If nodes are at same position, we order them by value.

ex:



→ op [— [1,5,6], —]

A)
```
vector<vector<int>> verticalTraversal(Node* root){
    vector<vector<int>> ans;
    if(root==NULL) return ans;
    queue<pair<Node*, pair<int,int>>> q;        // stores vertical & horizontal distance
    map<int, map<int, multiset<int>>> nodes;    // to store nodes sorted by their vertical & horizontal distance
    q.push({root, {0,0}});
    while(!q.empty()){
        auto p=q.front();
        q.pop();
        Node* node=p.first;
        int x=p.second.first;   -> horizontal distance
        int y=p.second.second;  -> Vertical distance
        nodes[x][y].insert(node->val);
        if(node->left!=NULL) q.push({node->left, {x-1, y+1}});
        if(node->right!=NULL) q.push({node->right, {x+1, y+1}});
    }
    for(auto p: nodes){
        vector<int> col;
        for(auto q: p.second){
            for(auto val: q.second){
                col.push_back(val);
            }
        }
        ans.push_back(col);
    }
    return ans;
}
```

$T_c = O(N * \log 2N * \log 2N * \log 2N)$ (no. of nodes)

multiset → to insert overlapping nodes at specific vertical & horizontal levels

map → insertion & retrieval

$S = O(N + N/2)$ (map) (queue for BFS)

Q) REPEATED SUBSTRING PATTERN → (refer KMP Algo. in front of book)

Check if string 's' can be constructed by taking a substring of it & appending multiple copies of the substring together

ex: "aba" → false, "a" → false, "ababba" → false, "abaabaabaab" → true, "abab" → true

A)
```
bool repeatedSubstringPattern(string s){
    string t=s;
    for(int i=0; i<s.size()-1; i++){   -> Till s.size()-1, because we don't want full string to get rotated (it will return true, in that case, so avoiding)
        t+=t[0];
        t.erase(0,1);   -> (From 0th index, remove ele.s of size==1)
        if(t==s) return true;
    }
    return false;
}
```

$T_c = O(n^2)$, $S_c = O(n)$

* KMP algo. → for $T_c = O(n)$

Q) THE CELEBRITY PROBLEM

A 'n×n' matrix M, represents people at party such that if ele. of row 'i' & column 'j' is set to '1', it means i^{th} person knows j^{th} person. Here M[i][i] = 0 always. Celebrity is all other n-1 people know about him but he does not know any of them. Find celebrity?

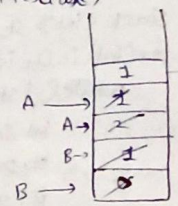
ex. N=3, M = {{0,1,0}, {0,0,0}, {0,1,0}} → ans: 1 (0th & 2nd person knows 1 ↓ : Celebrity)

A) Brute force to find: → check if that whole row = 0 & that col. = '1' except diagonal ele.s

↳ $T_c = O(n^2)$

Optimal:

(ex. → in textrace)



exp. Algo: → put all ele.s inside stack
→ until stack size != 1 → A → s.top() → s.pop()
↳ B → s.top() → s.pop()
→ if(A knows B) → A ∝, 'B' → push in stack
→ if(B knows A) → B ∝, 'A' → push in stack
→ Single ele. left = potential candidate
→ Verify that ele. → celebrity knows no-one → row check all 0's
↳ everyone knows celebrity → col, all '1' except digonal

Code:
```
bool knows(vector<vector<int>>& M, int a, int b, int n){
    if(M[a][b]==1) return true;
    else return false;
}
int celebrity(vector<vector<int>>& M, int n){
    stack<int> st;
    for(int i=0; i<n, i++) st.push(i);   -> pushing all ele.s in stack
    while(st.size()>1){   -> get 2 ele.s & compare them
        int a=st.top(); st.pop();
        int b=st.top(); st.pop();
        if(knows(M,a,b,n)) st.push(b);
        else st.push(a)
    }
    int ans=st.top();   -> potential celeb.
    int zeroCount=0, oneCount=0;
    for(int i=0; i<n; i++){
        if(M[ans][i]==0) zeroCount++;
    }
    if(zeroCount!=n) return -1;
    for(int i=0; i<n; i++){
        if(M[i][ans]==1) oneCount++;
    }
    if(oneCount!=n-1) return -1;
    return ans;
}
```

$T_c = O(n)$

$S_c = O(n)$

Q) MAXIMUM OF MINIMUM FOR EVERY WINDOW SIZE

Given array of 'N' integers, find max of min for every window size.

ex: arr = [1,2,3,4]

Window size 1 → min(1), min(2), min(3), min(4) = 1,2,3,4 → max = 4
Window size 2 → min(1,2), min(2,3), min(3,4) = 1,2,3 → max = 3
Window size 3 → min(1,2,3), min(2,3,4) = 1,2 → max = 2
Window size 4 → min(1,2,3,4) = 1 → max = 1

∴ op: 4 3 2 1

- $1 \le T \le 10^5$
- $1 \le N \le 10^5$
- $-10^9 \le arr[i] \le 10^9$

A) Brute force → traversing all windows & checking → $T_c = O(N^3)$

Better → Using deque → $T_c = O(N^2)$

Optimal: → $T_c = O(N)$, $S_c = O(N)$

```
vector<int> maxMinWindow(vector<int> a, int n){
    vector<int> ans(n, INT_MIN); //initialize 'ans' with size 'n' & all elements set to INT_MIN
    stack<int> st;
    for(int i=0; i<n; i++){
        while(!st.empty() and a[st.top()] > a[i]){
            int index = st.top();
            st.pop();
            if(st.empty()){ // NSE - PSE - 1 = range. If stack empty -> PSE = -1
                int range = i - (-1) - 1;
                ans[range-1] = max(ans[range-1], a[index]);
            }
            else{
                int range = i - st.top() - 1;
                ans[range-1] = max(ans[range-1], a[index]);
            }
        }
        st.push(i);
    }
    while(!st.empty()){
        int index = st.top();
        st.pop();
        if(st.empty()){
            int range = n - (-1) - 1;
            ans[range-1] = max(ans[range-1], a[index]);
        }
        else{
            int range = n - st.top() - 1;
            ans[range-1] = max(ans[range-1], a[index]);
        }
    }
    for(int i = n-2; i>=0; i--){
        ans[i] = max(ans[i], ans[i+1]);
    }
    return ans;
}
```

ex: index = 3 → range = 4-2-1 = 0 | = 4-1-1 = 2

st: [3, 2, 1, 0]

ans: [4 | 3 | 2 | 1] (0 1 2 3)

(All initialized with INT_MIN)

- Last for loop used when, ex

[10 | 20 | 15 | 30 | 10 | 70 | 30]

Let ans: [70 | 30 | 15 | 0 | 0 | 0 | 10] (0 1 2 3 4 5 6)

max(0, 10) = 10
max(0, 10) = 10

Last 'for-loop'

we are getting 0 as we are updating only particular index in 'ans'.

- Do dry run, if not understood

<Refer KMP Algorithm />

Q) COMPARE VERSION NUMBERS

Compare 2 version strings, ver1 & ver2. A version string consists of revisions seperated by dots '.'. The value of the revision is its integer conversion ignoring leading zeroes. To compare version strings, compare their revision values in left-to-right order. If one of the version strings has fewer revisions, treat the missing revision values as 0.

return: → '-1' : ver1 < ver2 , '1' : ver1 > ver2 , else '0'

ex: v1 = "1.2", v2 = "1.10"
op: -1
exp: v1's 2nd revision 2 & v2's 2nd revision '10'. 2 < 10 → ∴ v1 < v2

ex: v1 = "1.01", v2 = "1.001"
op: 0
exp: ignoring leading zeroes → 01 = 1, 001 = 1

ex: v1 = "1.0", v2 = "1.0.0.0"
op: 0
exp: v1 has less revisions, which means every missing revision = '0'.

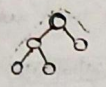
A)
```
int compareVersion(string v1, string v2){
    int i=0, j=0, v1Len = v1.size(), v2Len = v2.size(), n1, n2;
    while(i<v1Len or j<v2Len){
        n1 = 0; n2 = 0;
        while(i<v1Len and v1[i]!='.'){
            n1 = n1*10 + (v1[i]-'0');
            i++;
        }
        while(j<v2Len and v2[j]!='.'){
            n2 = n2*10 + (v2[j]-'0');
            j++;
        }
        if(n1<n2) return -1;
        else if(n1>n2) return 1;
        i++; j++;
    }
    return 0;
}
```

$T_c = O(m+n)$

$S_c = O(1)$

Q) DIAMETER OF BINARY TREE

Calc. dia. of B-tree. Diameter is length of longest path between any 2 nodes in a tree. {path may not pass via root}. Length = edges b/w nodes

ex:  o/p: 3

ex:  o/p: 6

A) Brute force: (Algo.) → $T_c = O(n^2)$

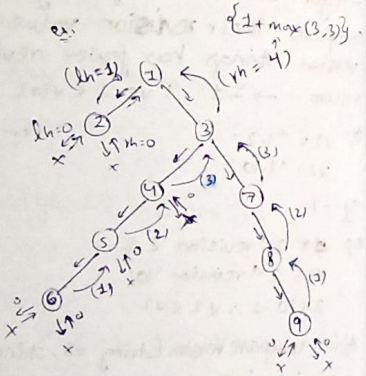
```
findMax(node) {                    // (root)
    if(root == NULL) return;
    lh = depth(node->left);
    rh = depth(node->right);
    maxi = max(maxi, lh+rh);
    findMax(node->left);
    findMax(node->right);
}
```

- 'Depth' function = height of binary tree (written in "Binary Trees - 1")
- Approach:
  - → length of longest path passing via given node = left depth + right depth
  - → If we have N nodes, then we find 'lh+rh' for every given node. So we can say max of (lh+rh) is the diameter of any given B-Tree.

Optimal Soln.: → $T_c = O(N)$, $S_c = O(1)$, $O(H)$ → auxiliary stack space

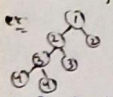
```
int depth(Node* root, int& diameter){
    if(root == NULL) return 0;
    int lh = depth(root->left, diameter);
    int rh = depth(root->right, diameter);
    diameter = max(diameter, lh+rh);
    return max(lh, rh)+1;
}

int diameterOfBinaryTree(Node* root){
    int diameter = 0;
    depth(root, diameter);
    return diameter;
}
```

ex: 

Q) BALANCED BINARY TREE

Check if tree is height-balanced. (depth of 2 subtrees of every node never differs by more than 1)

ex:  o/p: false

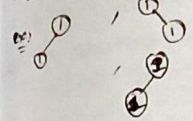
A) Brute force & optimal → similar to above answer.] do dry run of below code for better understanding

Optimal → $T_c = O(N)$, $S_c = O(1)$, $O(H)$ → auxiliary stack space

```
int check(Node* root){
    if(root == NULL) return 0;
    int lh = check(root->left);
    int rh = check(root->right);
    if(lh == -1 or rh == -1) return -1;
    if(abs(lh-rh) > 1) return -1;
    return max(lh, rh)+1;
}

bool isBalanced(Node* root){
    int ans = check(root);
    if(ans != -1) return true;
    return false;
}
```

Q) SAME TREE · Check whether 2 B-trees are same or not (structure & value)

ex:  → o/p: false

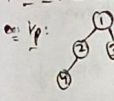
ex: → o/p: true

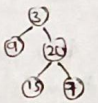
→ $T_c = O(p+q)$, $S_c = O(1)$, $O(H)$ → aux. space

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

Also, tried storing both trees in inorder traversed vector, but it will give wrong answer in 1st ex. above, so this method fails

Q) ZIG ZAG TRAVERSAL OF BINARY TREE

ex:  o/p: [[1], [3,2], [4,5]]

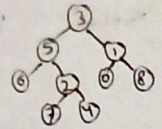
 o/p: [[3], [20,9], [15,7]]

A)
```
vector<vector<int>> zigzagLevelOrder(Node* root){
    vector<vector<int>> ans;
    if(root == NULL) return ans;
    queue<Node*> q;
    q.push(root);
    bool check = true;
    while(!q.empty()){
        int nodesAtCurrentLevel = q.size();
        vector<int> temp;
        while(nodesAtCurrentLevel--){
            Node* curr = q.front();
            q.pop();
            temp.push_back(curr->val);
            if(curr->right != NULL) q.push(curr->right);
            if(curr->left != NULL) q.push(curr->left);
        }
        if(check){
            reverse(temp.begin(), temp.end());
            ans.push_back(temp);
            check = false;
        }
        else{
            ans.push_back(temp);
            check = true;
        }
    }
    return ans;
}
```

$T_c = O(N)$

$S_c = O(n)$

Q) LOWEST COMMON ANCESTOR OF A BINARY TREE



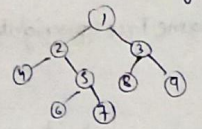
p=5, q=1 → ans: 3

p=5, q=4 → ans: 5

A) Brute force: → $T_c = O(N)$, $S_c = O(N)$

- Traverse the tree & maintain 2 vectors where each vector stores the path of 'p' & 'q' from root.
- Now compare vector of 'p' & 'q' & see the index where both values are diff. → its prev index = answer

ex.

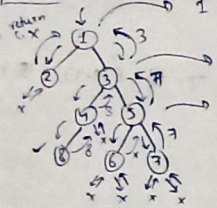


node = 4 → [1, 2, 4]

node = 7 → [1, 2, 5, 7]

diff. → so ans = prev index

Optimal: → $T_c = O(N)$, $S_c = O(H)$ → aux. stack space



'1' has null & '3'. So choose other of null i.e. 3 → ans: 3

'3' has '8' & '7' as returned values, so we return '3'.

'5' has 'null' & '7' as returned values. If one of them is null, return the other values

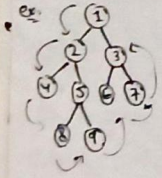
Code:

```
Node* LCA(Node* root, Node* p, Node* q){
    if(root==NULL or root==p or root==q) return root;
    Node* left = LCA(root->left, p, q);
    Node* right = LCA(root->right, p, q);
    if(left==NULL) return right;
    else if(right==NULL) return left;
    return root; → if left & right → not null, we found our result
}
```

Q) BOUNDARY TRAVERSAL

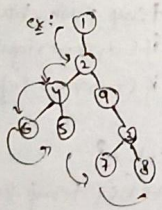
Find boundary traversal of a tree. (in anticlockwise direction)

ex:



→ 1 2 4 8 9 6 7 3

ex:



→ 1 2 4 6 5 7 8

A) What we can do → traverse leftmost part

→ traverse leaf nodes

→ traverse rightmost part & reverse it.

Code: → $T_c = O(N)$, $S_c = O(N)$ → if skewed tree

```
bool isLeaf(CNode* root){
    if(root->left==NULL and root->right==NULL) return true;
    return false;
}
void addLeftBoundary(CNode* root, vector<int> &res){
    Node* curr = root->left;
    while(curr!=NULL){
        if(!isLeaf(curr)) res.push_back(curr->data);
        if(curr->left!=NULL) curr = curr->left;
        else curr = curr->right;
    }
}
void addLeaves(Node* root, vector<int> &res){
    if(isLeaf(root)){
        res.push_back(root->data);
        return;
    }
    if(root->left!=NULL) addLeaves(root->left, res);
    if(root->right!=NULL) addLeaves(root->right, res);
}
void addRightBoundary(Node* root, vector<int> &res){
    Node* curr = root->right;
    vector<int> temp;
    while(curr!=NULL){
        if(!isLeaf(curr)) temp.push_back(curr->data);
        if(curr->right!=NULL) curr = curr->right;
        else curr = curr->left;
    }
    for(int i=temp.size()-1; i>=0; i--) res.push_back(temp[i]);
}
vector<int> boundary(Node* root){
    vector<int> res;
    if(root==NULL) return res;
    if(!isLeaf(root)) res.push_back(root->data);
    addLeftBoundary(root, res);
    addLeaves(root, res);
    addRightBoundary(root, res);
    return res;
}
```

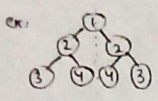
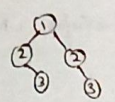
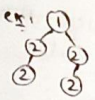
traversing left boundary

adding leaf nodes

traversing right boundary

Q) SYMMETRIC TREE

Given root of B-tree, check whether it's a mirror of itself.

ex:  op: true , ex:  op: false , ex:  op: false

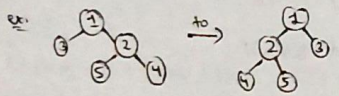
A) Code: → $T_c = O(N)$, $S_c = O(N)$ ↳ aux. stack space

```
bool isSymmetricHelper(Node* l, Node* r){
    if(l==NULL or r==NULL) return l==r;   // checks both condition whether one is null or both are null.
    if(l->val != r->val) return false;
    return isSymmetricHelper(l->left, r->right) and isSymmetricHelper(l->right, r->left);
}
bool isSymmetric(Node* root){
    if(root == NULL) return true;
    return isSymmetricHelper(root->left, root->right);
}
```

Also, tried storing values in array → (inorder traversal), but it will give error on 3rd ex., so this soln. is not correct.

Q) MIRROR TREE

Convert B-tree to its mirror;

ex: 

A) → $T_c = O(N)$, $S_c = O(N)$ ↳ aux. stack space

```
void mirror(Node* root){
    if(root==NULL) return;
    else{
        Node* temp;
        mirror(root->left);
        mirror(root->right);
        temp = root->left;
        root->left = root->right;
        root->right = temp;
    }
}
```

- If we are at 'current' node, then we are swapping its child nodes & returning.
- Following above step recursively.

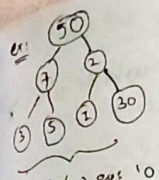
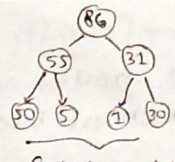
Q) CHECK FOR CHILDREN SUM PROPERTY

a) Check whether all nodes of a B-tree have the value equal to the sum of their child nodes value. Return 1 if true, else return 0.

b) Convert the B-tree to follow child-sum property.

(states that in a B-tree, every node, the sum of its children's values (if they exist) should be equal to the node's value.)

Note:
- → Node values can be ↑, but can't be decreased
- → A value for NULL = '0'
- → We can't change str. of B-tree

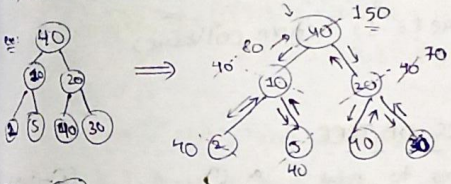
ex:  ⟷ (ex: (b)) → 

for 'a' → op: '0' for 'a' → op: '1'

A) a) `int isSumProperty(Node* root){` → $T_c = O(N)$, $S_c = O(N)$

```
    if(root == NULL) return 1;
    if(root->left == NULL and root->right == NULL) return 1;   // Leaf Nodes
    int left_data = (root->left != NULL) ? root->left->data : 0;
    int right_data = (root->right != NULL) ? root->right->data : 0;
    if(root->data == left_data + right_data){
        if(isSumProperty(root->left) and isSumProperty(root->right)){
            return 1;
        }
    }
    return 0;
}
```

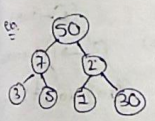
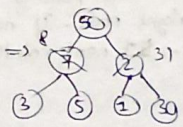
b) Here we have to convert. Let's understand via example:

ex: 

{10+20 = 30 < 40} → if addn. is lesser, assign curr. value

{2+5 = 7 < 40}

{30+40 = 70 > 40}

ex:  ⇒ If we try to go from bottom ⇒ 

{31+8 < 50} ↳ we can't reduce 50 value, so we do above method

Code: → $T_c = O(N)$, $S_c = O(H)$ i.e. $O(N)$ → worst case

```
void changeTree(Node* root){
    if(root==NULL) return;
    int child=0;
    if(root->left != NULL) child += root->left->val;   // Calculating the sum of values of the left & right children.
    if(root->right != NULL) child += root->right->val;
    if(child > root->data) root->data = child;   // Compare sum of children with curr. node's value & update
    else{ // if sum->smaller, update child with current node's value
        if(root->left != NULL) root->left->val = root->val;
        if(root->right != NULL) root->right->val = root->val;
    }
    changeTree(root->left);
    changeTree(root->right);
    int tot=0; // Calculating total sum of the values of the left & right children
    if(root->left != NULL) tot += root->left->val;
    if(root->right != NULL) tot += root->right->val;
    if(root->left != NULL or root->right != NULL) root->val = tot;
}
```

Q) FLOOR AND CEIL IN BST → (consists of only +ve no.'s)

Given a value 'x', a) find greatest value node which is <= x (right of BST)
b) find closest integer >= 'x'.

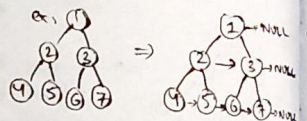
(FLOOR) → $T_c = O(N)$, $S_c = O(N)$ → aux. stack space

A) a)
```
int floorInBST(Node* root, int x){
    if(root == NULL) return -1;
    if(root->val == x) return root->val;
    if(root->val > x) return floorInBST(root->left, x);
    int floorValue = floorInBST(root->right, x);
    if(floorValue <= x and floorValue != -1) return floorValue;
    return root->val;
}
```
if root->val < x → potential floor element

b)
```
int findCeil(Node* root, int x){
    if(root == NULL) return -1;
    if(root->data == x) return root->data;
    if(root->data < x) return findCeil(root->right, x);
    int ceilValue = findCeil(root->left, x);
    if(ceilValue >= x and ceilValue != -1) return ceilValue;
    return root->data;
}
```
height of tree
$T_c = O(N)$
$S_c = O(N)$
if root->data > x → potential ceil value

Q) POPULATE NEXT RIGHT POINTERS OF TREE

```
struct Node{
    int val;
    Node* left;
    Node* right;
    Node* next;
}
```
Populate each next pointer to point its next right node. If no next right node → set next pointer to NULL



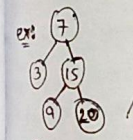
A)
```
Node* connect(Node* root){
    if(root == NULL) return NULL;
    queue<Node*> q;
    q.push(root);
    while(!q.empty()){
        int nodesAtCurrentLevel = q.size();
        Node* prev = NULL;
        while(nodesAtCurrentLevel--){
            Node* curr = q.front();
            q.pop();
            if(prev != NULL) prev->next = curr;
            prev = curr;
            if(curr->left) q.push(curr->left);
            if(curr->right) q.push(curr->right);
        }
        prev->next = NULL;
    }
    return root;
}
```
$T_c = O(N)$
$S_c = O(N)$

Q) K^th LARGEST ELEMENT IN BST → $T_c = O(N)$, $S_c = O(1)$

```
void inorder(Node* root, int& cnt, int K, int n, int& KLargest){
    if(root == NULL) return;
    inorder(root->left, cnt, K, n, KLargest);
    cnt++;
    if(cnt == n-K+1){
        KLargest = root->data;
        return;
    }
    inorder(root->right, cnt, K, n, KLargest);
}
int KthLargest(Node* root, int K){
    int cnt = 0, KLargest = 0, n = 0;
    totalNodes(root, n);
    inorder(root, cnt, K, n, KLargest);
    return KLargest;
}
```
```
void totalNodes(Node* root, int& n){
    if(root == NULL) return;
    totalNodes(root->left, n);
    n++;
    totalNodes(root->right, n);
}
```
• For K^th Smallest Element
(traverse & check if cnt == K → KLargest = root->data)

Q) BINARY SEARCH TREE ITERATOR

boolean hasNext() → true if num exists in traversal to right of pointer

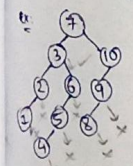
ex: 
aka: [[[7,3,15,null,null,9,20]],[],[],[],[],[],[],[]]

```
BSTIterator object;
BSTIterator* obj = new BSTIterator(root);
int param_1 = obj->next();
bool param_2 = obj->hasNext();
```

ip: ["BSTIterator","next","next","hasNext","next","hasNext","next","hasNext","next","hasNext"]
op: [null,3,7,true,9,true,15,true,20,false]

int next() → moved pointer to right, then returns no. at pointer.

A) Approach → push all elements in stack in this manner (Leftmost node in stack → if next return this & pop this ele., now add it's right subtree & in that left subtree. Repeat)

ex: 

Stack: 8, 9, 10, 4, 5, 6, 1, 2, 3, 7

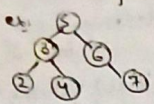
BstIterator(7)
next → 1
next → 2
next → 3
hasNext → true
next → 4
next → 5
next → 6
next → 7
next → 8
next → 9
next → 10

Code: → $S_c = O(H)$, $T_c = O(1)$

```
stack<Node*> st;
void pushAll(Node* node){
    while(node != NULL){
        st.push(node);
        node = node->left;
    }
}
BSTIterator(Node* root){
    pushAll(root);
}
int next(){
    Node* temp = st.top();
    st.pop();
    pushAll(temp->right);
    return temp->val;
}
bool hasNext(){
    return !st.empty();
}
```
Insert all the left elements

Q) 2 SUM - INPUT IS A BST

True → if 2 elements in BST, such that their sum = k, else false

ex: 

k=9, o/p: true

A) Using stack approach → 2 pointers → extreme left, extreme right
if sum > k → right--
else → left++
↳ $T_c = O(N)$, $S_c = O(H)$

Code

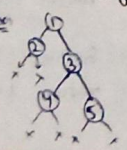
```
bool findTarget(Node* root, int k){
    if(root == NULL) return false;
    stack<Node*> st1, st2;
    Node* temp1 = root, *temp2 = root;
    while(temp1){
        st1.push(temp1);
        temp1 = temp1->left;
    }
    while(temp2){
        st2.push(temp2);
        temp2 = temp2->right;
    }
    while(!st1.empty() and !st2.empty() and st1.top() != st2.top()){
        Node* leftNode = st1.top(), *rightNode = st2.top();
        int sum = leftNode->val + rightNode->val;
        if(sum == k) return true;
        else if(sum < k){
            st1.pop();
            temp1 = leftNode->right;
            while(temp1){
                st1.push(temp1);
                temp1 = temp1->left;
            }
        }
        else{ // (sum > k)
            st2.pop();
            temp2 = rightNode->left;
            while(temp2){
                st2.push(temp2);
                temp2 = temp2->right;
            }
        }
    }
    return false;
}
```

Q) SERIALIZE & DESERIALIZE BINARY TREE

Given a B-tree, design algo. to seria. & de-seria. it. Serialisation = process of translating a data structure into a format that can be stored or transmitted & reconstructed later. Opposite operation, i.e. extracting data structure from stored info. = deserialisation. Ensure serialised tree can be deserialized to original tree. No restriction on how serialisation & deserialisation takes place.

↳ will be passed 'root' ↳ will be passed 'string'

A) We will use level order traversal.

ex: 

String: 1,2,3,#,#,4,5,#,#,#,# (# means NULL)

↳ Using queue & deserializing it.

Code → $T_c = O(N)$, $S_c = O(N)$

```
string serialize(Node* root){
    if(root == NULL) return "";
    string s = "";
    queue<Node*> q;
    q.push(root);
    while(!q.empty()){
        Node* curNode = q.front();
        q.pop();
        if(curNode == NULL) s.append("#,");
        else{
            s.append(to_string(curNode->val)+",");
            q.push(curNode->left);
            q.push(curNode->right);
        }
    }
    return s;
}
```

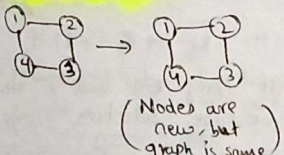
```
Node* deserialize(string data){
    if(data.size() == 0) return NULL;
    vector<string> nodes; string temp;
    for(int i=0; i<data.size(); i++){
        if(data[i] != ',') temp.push_back(data[i]);
        else{
            nodes.push_back(temp);
            temp.clear();
        }
    }
    if(temp.length() != 0) nodes.push_back(temp);
    Node* root = new Node(stoi(nodes[0]));
    queue<Node*> q; int i = 1;
    q.push(root);
    while(!q.empty()){
        Node* curr = q.front();
        q.pop();
        if(nodes[i] != "#"){
            Node* lRoot = new Node(stoi(nodes[i]));
            curr->left = lRoot;
            q.push(lRoot);
            i++;
        } else i++;
        if(nodes[i] != "#"){
            Node* rRoot = new Node(stoi(nodes[i]));
            curr->right = rRoot;
            q.push(rRoot);
            i++;
        } else i++;
    }
    return root;
}
```

Q) COUNT DISTINCT ELEMENTS IN EVERY WINDOW → (size = k)

ex: N=7, K=4, arr[] = {1,2,1,3,4,2,3}
o/p: {3,4,4,3}

A)
```
vector<int> distinct(vector<int>& a, int n, int k){
    vector<int> ans;
    unordered_map<int,int> mp;
    for(int i=0; i<k; i++) mp[v[i]]++;
    ans.push_back(mp.size());
    for(int i=k; i<n; i++){
        if(mp[v[i-k]] == 1) mp.erase(v[i-k]);
        else mp[v[i-k]]--;
        mp[v[i]]++;
        ans.push_back(mp.size());
    }
    return ans;
}
```

Q) CLONE GRAPH

ex: 

(Nodes are new, but graph is same)

```
class Node{
    int val;          // initial = 0
    List<Node> neighbours;   // initial = empty vector<Node>
}
```

Map → <old, new>

A)
```
Node* cloneGraph(Node* node){
    if(node == NULL) return NULL;
    unordered_map<Node*, Node*> mp;
    return cloneUtil(node, mp);
}
```

$T_c = O(V+E)$

$S_c = O(V+E)$

Code: (BU) → T_c = O(N*W), S_c = O(N*W)

```
int maxProfit(vector<int>& val, vector<int>& wt, int n, int maxWt){
    vector<vector<int>> dp;
    dp.resize(n+1, vector<int>(maxWt+1, 0));
    for(int i=wt[0]; i<maxWt; i++) dp[0][i] = val[0];
    for(int i=1; i<n, i++){
        for(int w=0; w<=maxWt; w++){
            int notTake = 0 + dp[i-1][w];
            int take = INT_MIN;
            if(wt[i]<=w) take = val[i] + dp[i-1][w-wt[i]];
            dp[i][w] = max(take, notTake);
        }
    }
    return dp[n-1][maxWt];
}
```

Draw table, for better clarity

Code: (BU) → Space Optimized → T_c = O(N*W), S_c = O(W)

```
int maxProfit(vector<int>& val, vector<int>& wt, int n, int maxWt){
    vector<int> prev(maxWt+1, 0);
    for(int W=wt[0]; W<=maxWt; W++) prev[W] = val[0];
    for(int idx=1; idx<n; idx++){
        for(int W=maxWt; W>=0; W--){
            int notTake = 0 + prev[W];
            int take = INT_MIN;
            if(wt[idx]<=W) take = val[idx] + prev[W-wt[idx]];
            prev[W] = max(take, notTake);
        }
    }
    return prev[maxWt];
}
```

Q) LONGEST_INCREASING_SUBSEQUENCE : return length of longest strictly ↑ subsequence in 'int' array.

Ans Soln. 1: Use DP, refer notes (2)

Soln. 2: ex: 2 5 (3) 7 11 9 10 13 6 8 9 10 12

- LIS array: 2 5 3 7 11 9 10 13 6 8 9 10 12 (indices 1 2 6 8 9 10 12)
- Traverse → add no. if it's less than no. in LIS array
  - → 3<5, so we change just higher val. than '3' i.e '5' here to '3'
  - → 9<11 → so change 11 to 9
  - → 6<13, 6<10, 6<9, 6<7 → So change '7' to 6
  - → So you can see that LIS array size is not decreasing, it's either same or increasing only.
- T_c = O(n log n) ; using 'lower_bound' → uses binary search 'log n' → T_c
- S_c = O(n)

```
int lengthOfLIS(vector<int>& nums){
    int n = nums.size();
    vector<int> v;
    v.push_back(nums[0]);
    for(int i=1; i<n; i++){
        if(v.back() < nums[i]) v.push_back(nums[i]);
        else{
            int idx = lower_bound(v.begin(), v.end(), nums[i]) - v.begin();
            v[idx] = nums[i];
        }
    }
    return v.size();
}
```

Q) MAXIMUM_SUM_INCREASING_SUBSEQUENCE → T_c = O(n²), S_c = O(n)

Find strictly increasing subsequence.

ex: N=5, arr[] = {1, 101, 2, 3, 100}, o/p: 106 → exp. {1, 2, 3, 100}

A) Code: (TD)

```
int msISHelper(vector<int>& arr, int n, int i, vector<int>& dp){
    if(dp[i] != -1) return dp[i];
    int max_sum = arr[i];
    for(int j=0; j<i; j++){
        if(arr[j]<arr[i]){
            max_sum = max(max_sum, arr[i] + msISHelper(arr, n, j, dp));
        }
    }
    return dp[i] = max_sum;
}
int maxSumIS(vector<int>& arr, int n){
    vector<int> dp(n, -1);
    int ans = INT_MIN;
    for(int i=0; i<n; i++) ans = max(ans, msISHelper(arr, n, i, dp));
    return ans;
}
```

Code: (BU)

```
int maxSumIS(vector<int>& arr, int n){
    int ans = arr[0];
    vector<int> dp(n);
    dp[0] = arr[0];
    for(int i=1; i<n; i++){
        dp[i] = arr[i];
        for(int j=0; j<i; j++){
            if(arr[j]<arr[i] and dp[i] < dp[j]+arr[i]){
                dp[i] = dp[j] + arr[i];
            }
        }
        ans = max(ans, dp[i]);
    }
    return ans;
}
```

dp[i] → subsequence ending at 'i' atleast includes arr[i] itself, that's why ini. val of dp[i] is set to that.

dp[i] → represents max. sum of increasing subsequence ending at index 'i'

```
Node* cloneUtil(Node* node, unordered_map<Node*, Node*>&mp){
    Node* newNode = new Node(node->val);
    mp[node] = newNode;
    for(auto ngbr : node->neighbours){
        //create neighbors clone
        if(mp.find(ngbr) == mp.end()) newNode->neighbours.push_back(
                                          cloneUtil(ngbr,mp));
        else (newNode->neighbors).push_back(mp[ngbr]);
    }
    return newNode;
}
```

BFS & DFS TRAVERSAL OF GRAPH

ex: V=5, E=4, adj = {{1,2,3},{},{4},{},{}} → o/p: BFS: 0 1 2 3 4
DFS: 0 1 2 4 3

A) BFS: → $T_c = O(V+E)$, $S_c = O(N)$

```
vector<int> bfs(int v, vector<vector<int>>& adj){
    vector<int> ans;
    vector<bool> track(v, false);
    queue<int> q;
    q.push(0);
    track[0] = true;
    while(!q.empty()){
        int curr = q.front();
        q.pop();
        ans.push_back(curr);
        for(auto neighbour : adj[curr]){
            if(!track[neighbour]){
                q.push(neighbour);
                track[neighbour] = true;
            }
        }
    }
    return ans;
}
```

DFS: → $T_c = O(V+E)$, $S_c = O(N)$

```
void dfs(vector<int>& ans, int node,
    vector<vector<int>>& adj, vector<bool>& track){
    track[node] = true;
    ans.push_back(node);
    for(int neighbour : adj[node]){
        if(!track[neighbour]){
            dfs(ans, neighbour, adj, track);
        }
    }
}
vector<int> dfsOfGraph(int v, vector<vector<int>>& adj)
{
    vector<int> ans;
    vector<bool> track(v, false);
    dfs(ans, 0, adj, track);
    return ans;
}
```

MAXIMUM PRODUCT SUBARRAY

Find subarray that has largest product in array. {Answer fits in 32 bits}

ex: [2,3,-2,4] → o/p: 6 → ex: [2,3]

A) Observation

- Only (+)ve's → all
- Even negatives → all
- Odd negatives
- Zeroes present

ex: 1 -2 3 4 -1 4 6 -1 4

So we ignore a negative & calc. pref. multiply & suff. multiply → ex: '-2'

{pref -2 suff} → max will be any one

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

(we find max of these both)

Similarly we consider break at '-1' & then calc. pref & suff. → max will be any one

Doing above at all negative & finding max

ex: -2 3 4 -1 0 -2 3 4 -1

pref → 1, -2, -6, -24, 24 {0} → so change pref = 1 (means we are starting new)

suff → 1, -1, 4, -24, 24 {0} → so change suff = 1 (means we are starting new)

Similarly we do for suffix, we calc. from end.

ans = max(pref, suff).

instead of iterating again, we can do → n-i+1 as we go i=0 to n

Code: → $T_c = O(N)$, $S_c = O(1)$

```
int maxProduct(vector<int>& nums){
    double pref = 1, suf = 1, n = nums.size(), ans = INT_MIN;
    for(int i=0; i<n; i++){
        if(pref==0) pref = 1;
        if(suf==0) suf = 1;
        pref = pref * nums[i];
        suf = suf * nums[n-i-1];
        ans = max(ans, max(pref, suff));
    }
    return ans;
}
```

0-1 KNAPSACK

You have 'N' items & you can either pick or not pick. Given items have weights array & values array. Given 'maxWeight', find max. profit you can have by picking weights such that their total is less than or = 'maxWeight'.

ex: val: 60 65 100, wt: 10 5 20, W = 25 → val/wt = 6, 13, 5 → Using greedy

(60/10) (65/5) (100/20)

65+60 = 135 & wt = 10+5 = 15 < 25

But actual answer → 100+65 = 165 {wt = 20+5 <= 25}

∴ 'Greedy' can't be applied because 'Uniformity' is not there i.e. "for less val, we have more wt", "for more value, we have less wt". So apply DP.

Code: (TD)

$T_c = O(N*W)$ → max N*W states are possible

$S_c = O(N*W) + O(N)$ → aux. stack space

```
int f(vector<int>& values, vector<int>& weights, int w, int i, vector<vector<int>>& dp){
    if(i==0){
        if(weights[0] <= w) return values[0];
        else return 0;
    }
    if(dp[i][w] != -1) return dp[i][w];
    int notTake = 0 + f(values, weights, w, i-1, dp);
    int take = INT_MIN;
    if(weights[i] <= w) take = values[i] + f(values, weights, w-weights[i], i-1, dp);
    return dp[i][w] = max(take, notTake);
}
int maxProfit(vector<int>& val, vector<int>& wt, int n, int maxWt){
    vector<vector<int>> dp;
    dp.resize(n+1, vector<int>(w+1, -1));
    return f(val, wt, w, n-1, dp);
}
```

Q) WORD BREAK : Given string 's' & a dict. of strings 'wordDict', return true if 's' can be segmented into a space-seperated sequence of one or more dictionary words. Note: same word in the dictionary can be used multiple times. → $T_c = O(n^2)$, $S_c = O(n)$

A) Similar to {Word-Break-2} (in this book only) → But if applied similar logic & checked size → would give TLE.

Code: (Top-Down)

```
bool wordB(string s, unordered_set<string>& wordSet, int start, vector<int>& dp){
    if(start==s.size()) return true;
    if(dp[start]!=-1) return dp[start];
    for(int i=start, i<s.size(); i++){
        string word = s.substr(start, i-start+1);
        if(wordSet.find(word)!=wordSet.end()){
            if(wordB(s, wordSet, i+1, dp)) return dp[start]=true;
        }
    }
    return dp[start]=false;
}

bool wordBreak(string s, vector<string>& wordDict){
    unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
    vector<int> dp(1005, -1);
    return wordB(s, wordSet, 0, dp);
}
```

this func. returns whether I can break the string from 'start' to end from the segment present in the wordSet

Code: (BU)

```
bool wordBreak(string s, vector<string>& wordDict){
    unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
    int n=s.size();
    vector<bool> dp(n+1, false);
    dp[n]=true; // empty substring at end of string can always be segmented.
    for(int i=n-1; i>=0; i--){
        for(int j=i+1; j<=n; j++){
            string word = s.substr(i, j-i);
            if(wordSet.find(word)!=wordSet.end()){
                if(dp[j]>0){
                    dp[i]=true;
                    break;
                }
            }
        }
    }
    return dp[0];
}
```

dp[i] → represents if s[i...n-1] can be segmented into words in the wordSet

Q) MINIMUM PATH SUM : find path from (0,0) → (m-1, n-1) which minimizes sum of all no.'s along its path.

{can move only ↓ or →}

A) Soln 1: Use DP, refer notes (2)

Soln 2: {Space Optimized} → $T_c = O(N*M)$, $S_c = O(m)$

Code:

```
int minPathSum(vector<vector<int>>& grid){
    int n=grid.size(), m=grid[0].size(); // n->rows, m->cols
    vector<int> prev(m,0), curr(m,0);
    prev[m-1]=grid[n-1][m-1];
    for(int i=m-2; i>=0; i--) prev[i]=grid[n-1][i]+prev[i+1];
    for(int i=n-2; i>=0; i--){
        curr[m-1]=grid[i][m-1]+prev[m-1];
        for(int j=m-2; j>=0; j--){
            curr[j]=grid[i][j]+min(curr[j+1], prev[j]);
        }
        swap(prev, curr);
    }
    return prev[0];
}
```

Explanation:

1	3	1
1	5	1
4	2	1

n=3, m=3

prev[2] = 1

prev:

7	3	1

(8) (7) (2)

curr:

i=1 → j=-1, 0 → | 8 | 7 | 2 |

i=0 → j=-1, 0 → | 7 | 3 | 1 | (swap)

ans: 7

'prev' stores min. cost to reach cell 'i,j' from [n-1][m-1]

'curr' stores

i=1 {n-3}
curr[2] = grid[1][2] + prev[2]
j=1
curr[1] = g[1][1] + min(curr[2], p[2])
j=0
→ swap(prev, curr);

As we go above we swap 'prev' & 'curr' so that my current row becomes previous & the 'next row' can be updated in 'curr'.

Q) EDIT DISTANCE

Total no. of operations req. to convert 'word1' to 'word2'
↳ insert, delete, update/replace

A)

	a			
b	0	1	2	3	4
	1				
	2				
	3				
	4				

Base Case: if a string is empty, then to get it to other word, we need to insert equal amount of chars.

```
int minDistance(string w1, string w2){
    int m = w1.size(), n = w2.size();
    vector<vector<int>> t(m+1, vector<int>(n+1));
    for(int i=0; i<m+1; i++) t[i][0] = i;
    for(int j=0; j<n+1; j++) t[0][j] = j;
    for(int i = 1; i<m+1; i++){
        for(int j=1; j<n+1; j++){
            if(w1[i-1] == w2[j-1]) t[i][j] = t[i-1][j-1];
            else t[i][j] = 1 + min(t[i-1][j-1], min(t[i-1][j], t[i][j-1]));
        }
    }
    return t[m][n];
}
```