{Knuth-Morris-Pratt} KMP Algorithm

- Used for pattern matching & finding occurences of a pattern in a string.

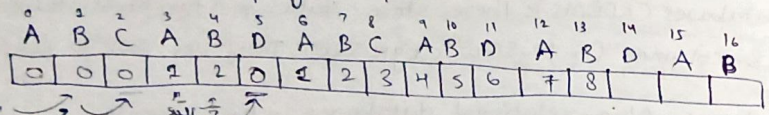
★) LONGEST PREFIX SUFFIX

Find length of longest proper prefix which is also a proper suffix.

Note: prefix & suffix can overlap but they should not be equal to entire string.

ex: s="abab" , ex: "abcabdabcabcabd"
op: 2 op: 6 -> i.e. "abcabd"

A) Explanation {with help of example}



(we maintain 2 variables)

↓

check if: pref != suff initially

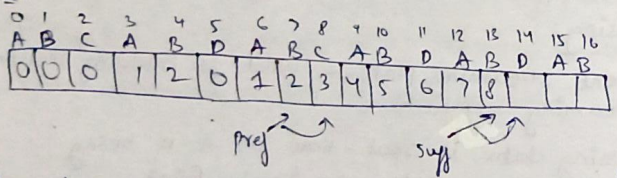
↳ Keep '0' in array

Case: when pref = 'C' & suff = 'D'

↓

Go to 'suff-1' no. & go to that index. (suff-1)

Here '2' at index 4' -> so going to index '2'.



Now when pref = 'C' & suff = 'D',

-> Go to 14-1 = 13th index -> check no. & go to that index i.e. 8

-> Since '8' i.e. 'C' != 'D' -> Go to 8-1 = 7

↓

Check no. & go to that index i.e 2

-> Since '2' i.e. 'C' != 'D' -> repeat above process

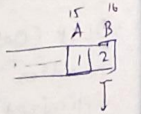
-> We will go before '0' index, so putting '0' at '14th' index

-> Reset pref to '0'.

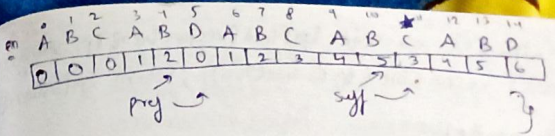
-> If ele found (or matched) -> then pref will begin from there

- If not matching, repeat above process, if goes to '0' index -> keep '0'.
  - Doing this to check if any pattern exists before or not.
  - Pref reset to '0'

∴



- $T_c = O(n)$ -> suffix always goes forward & prefix comes back sometimes only {so ignored} that too with long jumps
  $S_c = O(m)$



suff -> 'C' & pref = 'D'

-> going to prev i.e 'B' -> 5th index

-> not matching, so go to prev i.e. 'B' -> 2nd index

-> suff == pref -> so storing in array -> pref+1 & continuing the process...

Code:

```
int lps(string s){
    vector<int> lps(s.size(), 0);
    int pre = 0, suf = 1;
    while(suf < s.size()){
        //Match
        if(s[pre] == s[suf]){
            lps[suf] = pre+1;
            suf++; pre++;
        }
        else{ //Not matched
            if(pre == 0){
                lps[suf] = 0;
                suf++;
            }
            else{
                pre = lps[pre-1];
            }
        }
    }
    return lps[s.size()-1];
}
```

We can eliminate this line, as we initialized our vector 'lps' with 0

♦) FIND THE INDEX OF FIRST OCCURENCE IN A STRING

ex: haystack = "sat but sat" , needle = "sat" -> op: 0 -> occurs at 0th index.

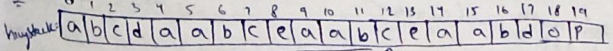
A) Brute force: O(mn) (T_c)

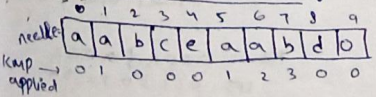
Algo:

```
n = haystack.size(), m = needle.size()
for(i = 0 to n-m){
    first = i, second = 0,
    while(second < m){
        if(haystack[first] != needle[second]) break;
        second++; first++;
    }
    if(second == m) return i;
}
return -1;
```

Optimal -> (KMP) -> $T_c = O(m+n)$

ex: first -> (since first != second -> so move on)





Not matching, so going 8-1 = 7 index no.

checking 3rd index -> 'c' = 'c'

matching -> so move on

If 'needle' there in 'haystack', then 'first' will stop as soon as 'second' is traversed completely. So answer -> "first - second", else -1.

Code

→ change : int lps(string s) ⇒ void lpsfind(vector<int>& lps, string s)
→ remove this lines: vector<int> lps(s.size(),0); , return lps[s.size()-1];

```
int strStr(string haystack, string needle){
    vector<int> lps(needle.size(),0);
    lpsfind(lps, needle);
    int first=0, second=0;
    while(first < haystack.size() and second < needle.size()){
        if(haystack[first] == needle[second]){
            first++; second++;
        }
        else{
            if(second==0) first++;
            else second = lps[second-1];
        }
    }
    if(second==needle.size()) return first-second;
    return -1;
}
```

Q) MIN. CHARS. NEEDED TO BE INSERTED IN BEGINNING TO MAKE IT PALINDROMIC

Operation allowed is to insert chars. at beginning of the string. Find how many min. chars. allowed/needed to be inserted in string to make it palindromic.

ex: "abc" → op: 2 → exp: cbabc , "aacecaaa" → op: 1 → aaacecaaa

A) ex: "aaaotcaakr"

We check (from start to end & find only "aaa" to be palindrome initially. This means all other chars other from this are to be repeated.

→ Finding "longest prefix palindrome" (LPP)

Another ex: ROORSP ⟷ (reversing) PSROOR

longest prefix palindrome ⟷ (both matching) longest suffix palindrome

(so can apply LPS)

- Now we make our string 's' as 's' + '$' + reverse(of 's')

ex: "aaaotcaakr"

aaaotcaakr$rkaactoaaa

LPS: 0 1 2 0 0 0 1 2 0 0 0 0 0 1 2 0 0 0 1 2 3

total no. of chars to be added is (s.size() - ans) elements from end ← (ans) ← tells this much size of LPP

ex: aaaa

LPS: 0 1 2 3

(if '$' not there)

aaaa$aaaa

LPS: 0 1 2 3 4 5 6 7

∴ ans = 4 - 7 = -3 ∝

∴ We introduce '$' to reset our LPS

Now: aaaa$aaaa

LPS: 0 1 2 3 0 1 2 3 4

∴ ans = 4 - 4 = 0 ✓ (already palindromic)

∴ Algo:
1) s
2) rev
3) st = s
4) st = rev
5) lps(s)

Code

```
int solve(string s){
    string rev=s;
    reverse(rev.begin(), rev.end());
    string x = s + '$' + rev;
    return s.size() - lps(x);
}
```

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"

op: 3 → exp: abcdabcdabcd

A)

```
bool lpsAns(string a, string b, vector<int>& lps){
    int first=0, second=0;
    while(first<a.size() and second<b.size()){
        if(a[first]==b[second]){
            first++; second++;
        }
        else{
            if(second==0) first++;
            else second = lps[second-1];
        }
    }
    if(second==b.size()) return true;
    return false;
}

int repeatedStringMatch(string a, string b){
    if(a==b) return 1;
    vector<int> lps(b.size(),0);
    lpsfind(lps, b);
    int n = b.size()/a.size();
    if(b.size()%a.size()!=0) n++;
    string repeatedA="";
    for(int i=0; i<n; i++) repeatedA+=a;
    if(lpsAns(repeatedA, b, lps)) return n;
    repeatedA+=a;
    if(lpsAns(repeatedA, b, lps)) return n+1;
    repeatedA+=a;
    if(lpsAns(repeatedA, b, lps)) return n+2;
    return -1;
}
```

→ check for 'b' in current repeatedA

"repeatedA + a"

Tc = O(m+n)
Sc = O(m+n)

checks if 'b' lies in 'a'

ex:
a = 'abcd'
b = 'cdabcdab'

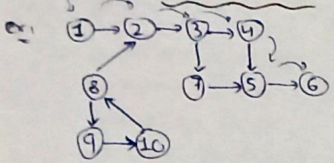
// b = pref. + (n*a) + suff.
= (cd) (abcd) (ab) → n+2

// b = pref + (n*a)
= (cd) (abcd) → n+1

// b = (n*a) + suff
= (abcd) (ab) → n+1

// b = (n*a)
= abcd → n

Detect Cycle in directed graph (DFS)



2 arrays:

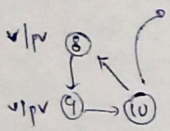
visited : 0 0 0 0 0 0 0 0 0 0 0

pathVisited : 0 0 0 0 0 0 0 0 0 0 0

- Keep visited & pathVisited as '1' as you traverse
- As you go back in DFS, make those 'pathVisted' back to '0' & keep visited as it is.

dfs(1) → dfs(2) → dfs(3) → dfs(4) → dfs(5) → dfs(6)
dfs(3) → dfs(7)

- Now when we get to '7', visited(7) = 1, pathVisited(7) = 1 → Now going to '5', we see visited already '1' but pathVisited = '0'.
  - → I will not call it cycle
  - → I will not go beyond '5' as it's repetition of traversal, so avoiding it
- Now dfs(8) → not going to '2' as it's 'vis' ✓



after 10 → we '8', vis. ✓ & pathVis ✓, so we say cycle is there (return true).

Code: → $T_c = O(V+E)$, $S_c = O(2N) \approx O(N)$

```
bool dfsCheck(int node, vector<vector<int>>& adj, vector<int>& vis, vector<int>& pathVis){
    vis[node] = 1;
    pathVis[node] = 1;
    for(auto it : adj[node]){                    // traversing adjacent nodes
        if(!vis[it]){
            if(dfsCheck(it, adj, vis, pathVis) == true) return true;
        }
        else if(pathVis[it]) return true;        // if node prev. visited & it is also in pathVis → cycle exists
    }
    pathVis[node] = 0;
    return false;
}

bool isCyclic(int v, vector<vector<int>>& adj){
    vector<int> vis(v,0);
    vector<int> pathVis(v,0);
    for(int i=0; i<v; i++){
        if(!vis[i]){
            if(dfsCheck(i, adj, vis, pathVis) == true) return true;
        }
    }
    return false;
}
```

For undirected graph:

```
int parent;
(...,node)
(it != parent)...
(...,-1)
```

X (remove)

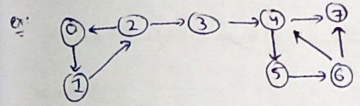
Now with these minor changes, we can find cycle in undirected graph as well.

Kosaraju's Algorithm

(Used to find total SCC) → strongly Connected Comp.

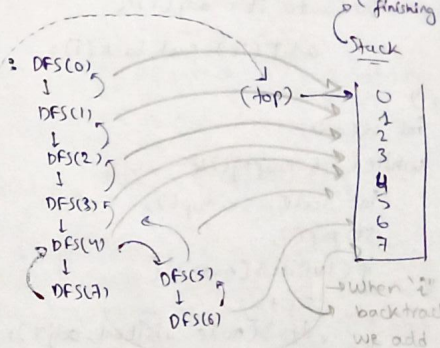
Strongly Connected Component:
In a directed graph, it's subset of vertices 'C' such that every pair of vertices u, v ∈ C, there exists both a path from u → v & a path from v → u.

Algo.: → Sort all the edges according to finishing time.
→ Reverse the graph
→ Do a DFS

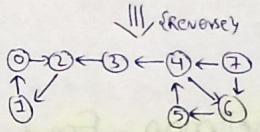


SCC : 0 1 2 / 3 / 4 5 6 / 7 — total → 4

Finishing Time : order in which vertices finish processing the DFS traversal phases.



(Reverse)



(DFS)

DFS(0)	DFS(3)	DFS(4)	DFS(7)
↓	↓↑	↓	↓
DFS(1)	2	DFS(6)	4, 6
↓	↳ (already visited)	↓	↳ (visited already)
DFS(2)		DFS(5)	

Ele. which is finishing last = 0.
∴ The 1st SCC will be having '0' so we take out '0'.
That's why sorting them based on finishing time was imp., because if we sort them, we know the 1st SCC & then we can go to 2nd SCC, ... etc.

∴ total DFS calls = 4
∴ SCC = 4

Code: → $T_c = O(V+E)$, $S_c = O(V+E)$

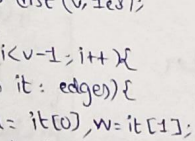
```
void dfs(int src, vector<int>& visited, vector<vector<int>>& adj, stack<int>& st){
    visited[src] = 1;
    for(auto it : adj[src]){
        if(!visited[it]) dfs(it, visited, adj, st);   // If neighbour not visited, then call DFS
    }
    st.push(src);
}

void dfs3(int src, vector<int>& visited, vector<vector<int>>& adjT){
    visited[src] = 1;
    for(auto it : adjT[src]){
        if(!visited[it]) dfs3(it, visited, adjT);
    }
}
```

```
int Kosaraju(int v, vector<vector<int>>& adj){
    vector<int> visited(v,0);                          } Sc = O(2V)
    stack<int> st;
    for(int i=0; i<v; i++){                  → Tc = O(V+E)
        if(!visited[i]) dfs(i, visited, adj, st);      } if not visited, call DFS
    }
    vector<vector<int>> adjT(v, vector<int>());        → Sc = O(V+E)
    for(int i=0; i<v; i++){                  → Tc = O(V+E)
        visited[i]=0;
        for(auto it: adj[i]){                          } Reversing the graph
            // i->it, it->i
            adjT[it].push_back(i);
        }
    }
    int scc=0;
    while(!st.empty()){                                } Tc = O(V+E)
        int node = st.top();
        st.pop();
        if(!visited[node]){
            scc++;
            dfs3(node, visited, adjT);
        }
    }
    return scc;
}
```

Bellman-Ford Algorithm

(used) → (applicable in DG only) → (shortest path algo)

- Only in directed graphs. If there is an undirected graph, convert it's undirected edge into 2 directed edges.
- Dijkstra Algo. generally fails when we have negative edges & if we have negative cycle, this algo. will give us TLE.
- Now bellman-ford algo. works well with negative edges & it can also help us identify negative cycle {path weight = 0} (of all edges, when summed)



(∴ cycle) {Path wt: 2-2-1 = -1}

- This algo. is also → Single source shortest path algo.

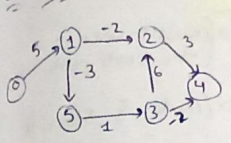
(*) Here edges can be in any order

Algo.: → Relax all edges 'n-1' times sequentially

→ Relaxation:

```
if(dist[u] + wt < dist[v]){
    dist[v] = dist[u] + wt;
}
```

ex: src = 0



dist[]: | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
0 1 2 3 4 5

(will store all my shortest path from src to all other nodes.)

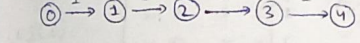
{2nd relax → 5 3 3 1 2 ; {1st relax → 5 3 6 2 ; So on ... (till 5th iteration)

(u, v, wt)

dist[3]+6 < dist[2]: (3, 2, 6)
dist[5]+1 < dist[3]: (5, 3, 1)
dist[0]+5 < dist[1]: (0, 1, 5)
dist[1]-3 < dist[4]: (1, 5, -3)
dist[1]-2 < dist[2]: (1, 2, -2)
dist[3]-2 < dist[4]: (3, 4, -2)
dist[2]+3 < dist[4]: (2, 4, 3)

Q1) Why 'n-1' iterations only?

A) Intuition:



(u,v, wt)
(3,4,1) : dist[3]+1 < dist[4]
(2,3,1) : dist[2]+1 < dist[3]
(1,2,1) : dist[1]+1 < dist[2]
(0,1,1) : dist[0]+1 < dist[1]

4th itr. → 1 2 3 4 ∴ My shortest path
3rd itr. → 1 2 3 ∞
2nd itr. → 1 2 ∞ ∞
1st itr. → 1 ∞ ∞ ∞

dist[] = | 0 | ∞ | ∞ | ∞ | ∞ |
0 1 2 3 4

Answer: Since in a graph of N nodes, in worst case, we will take N-1 edges to reach from the first to the last, thereby we iterate for N-1 iterations

Q2) How to detect negative cycles?

A) On nth iteration, the relaxation will be done & if the dist[] array gets reduced, we say negative cycle is present.

Intuition: On each iteration, value on index is getting reduced, & on (n-1)th iteration all values must be reduced. But if on Nth iteration, value is still getting reduced → ∴ (-ve cycle ✓

Code: → (Assumption: if vertex cant be reached from 's', mark 10^8 (or 1e8) → Return array consisting '-1' if negative cycle is present.)

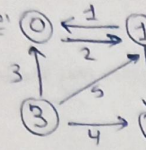
```
vector<int> bellman_ford(int v, int src, vector<vector<int>>& edges){
    vector<int> dist(v, 1e8);
    dist[src]=0;
    for(int i=0; i<v-1; i++){
        for(auto it: edges){
            int u=it[0], w=it[1];
            int wt=it[2];
            if(dist[u]!=1e8 and dist[u]+wt<dist[w]){
                dist[w]=dist[u]+wt;
            }
        }
    }
    for(auto it:edges){ // Nth relaxation to check negative cycle
        int u=it[0], w=it[1], wt=it[2];
        if(dist[u]!=1e8 and dist[u]+wt<dist[w]){
            return {-1};
        }
    }
    return dist;
}
```

Tc = O(V×E)
Sc = O(V)

Floyd - Warshall Algorithm

↳ (multisource shortest path algo.)

- It can also detect negative cycles also.
- Here we calculate node dist → from every vertex/node & store it.
- If we have : ① —5— ② → (convert) → ① ⇄ ② (5 each way)
  {undirected} {directed}
- We use adjacency matrix to store the graph.

ex: 

→ cost matrix }:

	0	1	2	3
0	0	2	∞	∞
1	1	0	3	∞
2	∞	∞	0	∞
3	3	5	4	0

move via vertex '0'

	0	1	2	3
0	0	2	∞	∞
1	1	0	3	∞
2	∞	∞	0	∞
3	3	5	4	0

ex [1][2] = [1][0] + [0][2]

(diag. = '0' as we are standing there only)

move via '1',

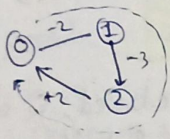
ex: [0][2] = [0][1] + [1][2] (via)

[3][0] = [3][1] + [1][0]

	0	1	2	3
0	0	2	5	∞
1	1	0	3	∞
2	∞	∞	0	∞
3	3	5	4	0

(We are comparing the min. of above present value & above calculated value)

Q) How to detect negative cycle?

A) ex: 

Cost(0) = -2-3+2 = -3

```
for(i=0 → n){
    if(cost[i][i]<0){
        "negative cycle"
    }
}
```

↳ {-1 <= matrix[i][j] <= 1000}

Code: → {changes done = in-place} ; $T_C = O(V^3)$, $S_C = O(V^2)$ ~ {matrix}

```
void shortest_distance(vector<vector<int>>& matrix){
    int n = matrix.size();
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            if(matrix[i][j] == -1) matrix[i][j] = 1e9;
            if(i==j) matrix[i][j] = 0;
        }
    }
    for(int k=0; k<n; k++){
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                matrix[i][j] = min(matrix[i][j],
                                   matrix[i][k]+matrix[k][j]);
            }
        }
    }
}
```

```
for(int i=0; i<n; i++){
    if(matrix[i][i]<0){
        cout<<"Negative cycle";
        break;
    }
}
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        if(matrix[i][j]==1e9)
            matrix[i][j]=-1;
    }
}
```

Questions Practiced In Notes 3

- Longest Subarray Sum with given sum K (+ & -) → $T_c = O(N \log N)$, $S_c = O(N)$

Ex: N=3, K=1, arr[] = {-1, 1, 1}
op: 3

⇒ Use unordered_map
⇒ keep adding & check if 'sum-K' is present in map

- Longest Subarray with given sum K (+'ve no.'s) → $T_c = O(2*N)$, $S_c = O(1)$

⇒ Use worm approach, for further optimization (above soln. is also correct) for this

⇒ 

i.e. keep adding ele.'s & if sum > k, then remove elements from beginning

- Kadane's algorithm → $T_c = O(N)$, $S_c = O(1)$

Find subarray with largest sum & return the sum

⇒ keep adding elements of array. If sum > prev.Sum, then update sum, else if it goes below 0, make sum as 0.

- All no.'s are repeated twice in an array except 2 no.'s. Find those 2 no.'s? → $T_c = O(N)$, $S_c = O(1)$

⇒ Maintain a variable 'numXor' which equals 'xor' of all elements.

⇒ Now to differentiate both no.'s, there will be atleast '1' bit different. So now we have to identify that bit.

↳ we do: → numXor ^ (numXor & (numXor - 1))

↳ removes the rightmost set bit

on doing xor, all bits will get cancelled except the rightmost set bit, as we are doing xor with 'numXor & (numXor-1)' which removes rightmost set bit.

⇒ Now running a loop in array & checking, if, any no. xor the 'rightbit' (storing above ans. in this variable) = 1, then doing it xor with another variable 'a' → (prev. initialised to '0'), else doing xor with another variable 'b' → (prev. initialised to '0').

⇒ So here we are creating 2 buckets (for analogy) & seperating the 2 unique no.'s in those buckets (since bucket is based on the rightmost set bit which is different in both).

• Without altering relative order of '+' & '-', return an array of alternately positive & negative values. → $T_c = O(N)$, $S_c = O(N)$

⇒ Make another vector 'ans'.

⇒ If arr[i] > 0 → Add in 'ans' & increase index by '2'. Similarly for arr[i] < 0.

• Longest Consecutive sequence length in a given unsorted array.

ex: [100, 4, 200, 1, 2, 3] → op: 4

⇒ Keep all elements in an unordered_set. → $T_c = O(N)$, $S_c = O(N)$

⇒ Now as you traverse the unordered_set, use a loop to check if (curr.ele. + 1) is present in set or not. If present, curr ele.++ & count++.

⇒ Outside this loop → ans = max(ans, count);

⇒ Note that first element of a sequence will not have any ele. lesser than it by 1. (← Hint: of starting the first loop of traversal of set).

• Total Subarrays whose sum = k → $T_c = O(N)$, $S_c = O(N)$

ex: [3, 1, 2, 4], K = 6 → op: 2

Note: Array ele.'s can be '+' & '-'

⇒ Use unordered map storing PrefixSum and it's count.

⇒ Initialise '0' with '1' in map, so that when '0' occurs we can increase our answer size by '1'.

⇒ Traverse the array & maintain 2 variables storing prefixSum & the sum required (let remove = prefixsum - k)

⇒ Add the frequency of 'remove' to 'ans' & increment the frequency of prefixSum by '1' in map.

⇒ Return ans.

• First and last occurence of a given no. in a sorted array

→ $T_c = O(\log N)$, $S_c = O(1)$

⇒ Use binary search

⇒ For first occurence, you need to reduce 'right' & for last occurence, you need to increase left.

• Prime factors of a number → $T_c = O(\sqrt{N})$, $S_c = O(\sqrt{N})$

⇒ Loop from 2 to sqrt(n) → {let 'i' = no. at any moment}

⇒ Check if 'n' is divisible by 'i'. If yes, the add that ele. in our 'ans' vector & keep dividing 'n' by 'i' unless until it's not divisible anymore.

• SIEVE OF ERATOSTHENES → $T_c = O(N \log(\log N))$, $S_c = O(N)$ {time will be to get, so ignore}

Prime factors till number 'N'

⇒ Initialise a vector → let 'prime' of size 'n+1' & value '1'

⇒ Loop from 2 to sqrt(n) & check if i^{th} index of prime = 1. If yes, then run another loop from i*i to n & increment by 'i' always & make j^{th} element in prime = 0.

⇒ Return the total no. of 1's in vector.

• Check if one string is rotation of another string (if KMP algo. was used)

⇒ Initialise new string = old string + old string → $T_c = O(m+2m) \approx O(m)$ {length of string}

⇒ Now find old string in this new string {in general} O(n+m) → $S_c = O(1)$

• Generate Binary Strings of length 'N' with no consecutive 1's in string

⇒ Initialise 'vector<strings>' & a 'temp' string.

⇒ Push 1st ele. i.e '0' & do recursive call, similarly now change that 1st ele. to '1' & do recursive call

⇒ In recursive call, if 'temp.size() == N' → then push in 'vector' else if → if prev char == '0' → do recursive calls adding '0' & '1' to temp & if prev char == '1' → do recursive calls adding '0' to temp.

$T_c = O(2^K)$, valid string grows exponen. → {F(K) = F(K-1) + F(K-2)}, $S_c = O(K \cdot 2^K)$ (length of 1 valid string) (total valid strings)

• Generate Parenthesis

ex: n = 3 → op: ["((()))", "(()())", "(())()", "()(())", "()()()"]

⇒ Maintain 3 variables → 'temp' string & append '(' initially

front = count of '(' & back = count of ')'

⇒ Do recursive calls & check if 'len' of temp == n*2, then add to vector, else if: front more than back → recursive call adding ')' & increasing back

else if: front less than n → recursive call adding '(' & increasing front

$T_c = O(4^n/\sqrt{n})$ {Catalan no.'s}, $S_c = O(n \cdot (4^n/\sqrt{n}))$ (size of a valid string) (total valid strings)

- Print all subsequences ~ power set → input = string → $T_c = O(2^n)$, $S_c = O(n)$
  ⇒ We need a vector<string> & a temp string. Do recursion
  ⇒ In recursion if curr index = s.size() → add to vector, else add curr string index char & recurse, and remove curr string index & recurse.

- Print all unique subsets → i/p: no. → (vector) → $T_c = O(2^n \cdot k)$, $S_c = O(2^n \cdot k)$
  ⇒ Storing result in 2D vector. Calling a recursion from index = 0
  ⇒ In recursion → We are adding our 'temp' vector to 'result' vector.
    ↳ Doing for loop from 'index' to 'n'
      ↳ if curr. & prev. element are same → continue.
      ↳ push current ele. & call recursion
      ↳ pop current ele.

- Total count of 'k' diff. characters → $T_c = O(n^2)$, $S_c = O(26)$
  ex: S = "aacfssa", k = 3 → o/p: 5 → exp: {aacf, acf, cfs, cfss, fssa}
  ⇒ Traversing the string & inside loop, we initialise a boolean array initially all false.
  ⇒ Using another loop inside 1st loop & here we traverse from curr index till end & check if 'char' position in boolean array is false, so we make it true & increment the count.
  ⇒ Also checking internally, whether count is = k, if yes so increment answer. If count surpasses given k, then break.

- Sum of beauty of all substrings of 's'
  (diff. btw most freq. occuring char. & least freq. occuring char) → $T_c = O(n^2)$, $S_c = O(26)$
  ⇒ Traverse from 0 to s.size().
    ↳ Initialise a vector, size 26 (as string → at max 'Z') → val = 0.
    ↳ Traverse from curr 'i' to end
      ↳ initialise 2 variables for 'min' & 'max'
      ↳ Increment vector's j^{th} ('char' - 'a') index by 1. (since char → so do "-'a'")
      ↳ traverse this vector
        ↳ Find min. & max. (excluding ele.s which are '0')
      ↳ Now add the diff. of max & min to another variable & return this variable at end.

- Find minimum in rotated sorted array → $T_c = O(\log n)$, $S_c = O(1)$
  ex: [3,4,5,1,2] → o/p: 1
  ⇒ Using binary search, we can atleast one-half is sorted.
  ⇒ If v[mid] < v[hi] : this means right part sorted & shift right to mid.
    else → change left to mid+1.
  ⇒ since right decreasing to 'mid' only, we know at which index our final ans. will be i.e 'left' → : return v[left]

- Find a peak element 2
  (element greater than all its neighbours : ←↑→↓) → $T_c = O(m \log n)$, $S_c = O(1)$

10	20	15
21	30	14
7	16	32

ex: o/p: (1,1)(or)(2,2)
  ⇒ We can use kinda binary search to help in traversal.
  ⇒ Using BS; we find max. row index at col. = 'mid' (you can do this in func.)
    If left & right of mid exists, then we equate it to element at maxRowIndex, left or right. → finding the largest ele. in that col. & returning its index (which denotes row)
      ↳ mid-1 ; ↳ mid+1
  ⇒ checking if ele. at maxRowIndex, mid > its left & right, if yes, then return, else if ele. at (,) is '<' left, then we do 'hi = mid-1', else ↑ing low to mid+1.

- Median of matrix (m,n), matrix = row-wise sorted
  $S_c = O(1)$ (add) $T_c = ((\log_2 10^9) \cdot n \log m)$ (search space → time by 'upper_bound' inside for loop)
  ⇒ We can use binary sort here also. Find the lowest & highest ele. in whole matrix {Hint: you just need to traverse 2 cols}
  ⇒ While, lowest < highest, we find 'mid'.
    ↳ Calculating no. of ele.s less than/equal to mid in whole matrix (↳ else: 'upper_bound')
    ↳ if freq. >= (m*n+1)/2
      ↳ potential answer → store it
      ↳ decrement highest to mid-1.
    ↳ else
      ↳ increment lowest to mid+1

- Sort 0's, 1's, 2's in an array → $T_c = O(n)$, $S_c = O(1)$
  ⇒ Consider : 0 to lo-1 → all 0's, lo to mid-1 → all 1's, mid to high → unsorted, high+1 to n-1 → all 2's
  ⇒ ∴ While → mid < hi → if ele. = 0 → swap mid & low
    ↳ ele = 1 → (↑) mid ; ele = 2 → swap mid & hi
  ⇒ You may initialise lo, mid = 0 & hi = n-1, & ↑(or)↓ as per if-else.

- Rotate ele.'s clockwise once in matrix

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

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

⇒ 4 variables → top, left, bottom, right are initialised

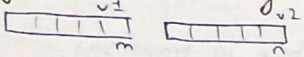
⇒ Loop until t < b & l < r

↳ Store m[t+1][l] in a var. 'prev'

↳ Now loop 4 times { l to r, t to b, r to l, b to t } → swap curr. ele. with prev

also increment/decrement accordingly

- Merge 2 sorted arrays without extra space

⇒ 

, total size = gap = ceil(len/2); (Sum of size of v1 & v2) (gives where the right index is starting)

⇒ while gap > 0 → left = 0 & right starts at left + gap.

↳ while right is less than len

↳ if → ((left < n) and (right >= n) and a[left] > b[right-n]) then swap both

↳ else if ((left >= n) and b[left-n] > b[right-n]) then swap both

↳ else → if (a[left] > a[right]) → then swap both

increment left & right

↳ if gap = 1 → break, else gap = ceil(gap/2);

- Merge all overlapping intervals

ex: [[1,3],[2,6],[8,10]] → [[1,6],[8,10]] → $T_c = O(N \log N) + O(N)$, $S_c = O(N)$

⇒ sorting the vector

⇒ for loop from 0 to n → (storing 'ans' in 2D vector)

↳ if ans.size = 0 → push arr[i];

↳ else if ans.back()[1] < arr[i][0] → push arr[i]

↳ else → ans.back()[1] = max(ans.back()[1], arr[i][1]);

(means ans.back()[1] > arr[i][0])

- Find duplicate in array → range: [1, n]

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

⇒ Use tortoise - hare algorithm

⇒ slow = nums[0] & fast = nums[0]

⇒ do → slow = nums[slow] & fast = nums[nums[fast]] while (slow != fast)

⇒ Cycle found → so fast = nums[0] & while (slow != fast) { slow = nums[slow]; fast = nums[fast]; }

⇒ return slow/fast.

- Add 2 no.'s, where no. → in form of reversed linked list

ex: (2→4→3) + (5→6→4) = 342 + 465 = 807 → : op: 7→0→8

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

⇒ Create a new Node → dummy & let Node* temp = dummy. & let carry = 0.

⇒ while (l1 != NULL / l2 != NULL / carry != 0)

↳ declare sum = 0

↳ add l1 → val & do next, similarly add l2 → val & do next (if ONLY they're not NULL)

↳ add carry to sum & do carry = sum/10;

↳ create new node with → 'sum%10' & then temp → next = (this new Node) & moving temp to next.

- Count Inversions → pair (a[i], a[j]) → a[i] > a[j] & i < j → arr → all distinct val.'s → find total inversion.

ip: 2 5 1 3 4

op: 4 → {(2,1), (5,1), (5,3), (5,4)}

⇒ Modifying merge sort, declare 'cnt' & add while doing recursion.

i.e. cnt += mergeSort(arr, lo, mid)

cnt += (" mid+1, hi)

cnt += merge(arr, lo, mid, hi);

Inside "mergesort" function.

in 'merge' function:

↳ same as 'mergeSort' application but 2 modifi-cations

↳ declare cnt = 0

↳ when arr[left] > arr[right]

↳ push into 'temp' vector

↳ cnt += (mid + 1 - left);

right++;

return cnt;

$T_c = O(N \log N)$

$S_c = O(N)$

- Find majority ele., that occurs more than n/2 / n/3 times in array.

⇒ For (n/2); {ans. only 1 ele.}

- traverse array

↳ if cnt = 0

↳ cnt = 1

↳ ele = v[i]

↳ else if (ele = v[i]) → cnt++

↳ else cnt--

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

⇒ Checking if curr. ele. is equal to most frequent ele., if yes, increase cnt, else decrease cnt.

⇒ if cnt = 0 → means another no. is better for answer, so changing ele = v[i] → (curr. ele.)

⇒ Traverse & check if v[i] == ele. → count1 ++;

↳ if count1 > n/2 → return ele, else -1

⇒ For (n/3); {ans. only 2 ele.}

⇒ traverse array

↳ if cnt1 = 0 & ele2 != v[i] → cnt1 = 1, ele1 = v[i]

↳ // for cnt2 & ele2

↳ else if (v[i] == ele1) cnt1++

↳ // for cnt2

↳ else → cnt1--, cnt2--;

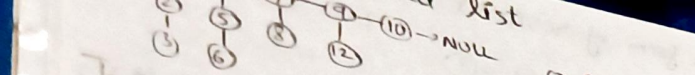
⇒ Now checking again

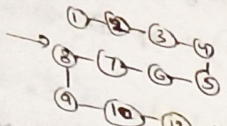
↳ for(0 to n) → if(== ele1) → c1++

↳ if(== ele2) → c2++;

↳ if (c1 > (n/3) & c2 > (n/3)) → answer

else {-1, -1}





'next' → points to next node in list
'child' → points to LL where curr_node is head
Return single sorted layer

⇒ Recursively going to end node {logic → merge last 2 lists & then similarly}
(head ⇒ NULL (ex) head → next ⇒ NULL → return head)
head → next = func_name (head → next);
return head;

⇒ head = mergeLL (head, head → next);
⇒ (create a temp node
⇒ while (a! = NULL & b! = NULL)
{ compare 'val' & then increment that LL accordingly & make to child
⇒ while (b! = NULL)
⇒ return res → child;

Node* temp = new Node(0);
Node* res = temp;

ex:
if (a → val < b → val){
temp → child = a;
temp = temp → child;
a = a → child;
}

- **Find N^th root of M**

ex: n=3, m=27 → o/p: 3

⇒ Can use binary search logic
⇒ lo = 1, hi = m, ans = 1
⇒ while lo less than hi → x = 1, mid = (lo+hi)/2
⇒ loop from 1 to n → x = x * mid; if (x > m) break;
⇒ if (x = m) → ans.✓
⇒ else if (x < m) → lo = mid + 1
⇒ else hi = mid - 1;

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

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

- **Search in rotated sorted array**

⇒ Using binary search ⇒ while (lo <= hi) & [mean v[mid]]
if (nums[mid] == tgt) return mid;
if (nums[lo] <= nums[mid]) → //left sorted
↳ check if 'tgt' lies btw 'lo' & 'mid' ↳ hi = mid - 1
else ↳ lo = mid + 1
else → check if 'tgt' btw 'mid' & 'hi' → lo = mid + 1
else → hi = mid - 1;

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

- **Median of 2 sorted arrays:**

ex: a1 → [1,3,4], a2 → [2,4] → o/p: 3

⇒ We will divide array in such a way that → l1 < r2 & l2 < r1, {l1 < r2 (by def. in array), l2 < r2}
ex: 1 3 4 (l1, r1) / 2 4 (l2, r2)
→ ∴ median = $\frac{\max(l_1, l_2) + \min(r_1, r_2)}{2}$

$T_c = O(\log(\min(n1, n2)))$
$S_c = O(1)$

⇒ lo = 0, hi = a1.size(); (shortest array among 2)
while (lo <= hi)
cut1 = $\left(\frac{lo+hi}{2}\right)$; , cut2 = $\left(\frac{n1+n2+1}{2}\right)$ − cut1;
l1 = cut1 == 0 ? INT_MIN : a[cut1 − 1], l2 = cut2 == 0 ? INT_MIN : b[cut2 − 1];
r1 = cut1 == n1 ? INT_MAX : a[cut1], r2 = cut2 == n2 ? INT_MAX : b[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)

- **No. of paths in grid to go from (0,0) to (m−1, n−1).** Can move only → or ↓
Obstacle = '1', space = '0'. Can't go via obstacle.

⇒ f(i,j) = f(i, j+1) + f(i+1, j)
↳ no. of ways of reaching bottom right from i,j by either going right/down

⇒ Top-down
if (i = n−1 & j = m−1) → 1
if (i >= n, j >= m, i < 0, j < 0) → 0
if (grid[i][j] == 1) → 0
if (dp[i][j] != −1) → dp[i][j]
dp[i][j] = f(i, j+1) + f(i+1, j);

$T_c = O(n \cdot m)$, $S_c = O(\max(n, m)) + $ 'dp'

⇒ Bottom-up
dp[n−1][m−1] = 1
set last row & last col of dp = grid i.e. if obs. → dp[i][j] = 1
else → dp[j][m−1] = dp[i+1][m−1] ((n−2) to 0)
dp[n−1][i] = dp[n−1][i+1] (i = m−2 to 0)

Now 2 for loops: → n−2 → 0, m−2 → 0
if grid[i][j] == 1 → dp[i][j] = 0
else dp[i][j] = dp[i+1][j] + dp[i][j+1]

$S_c = O(1)$ + 'dp'

- **Total reverse pairs in array** → (i, j) → 0 <= i < j < arr.size()
↳ a[i] > 2 * a[j]

⇒ Using merge sort → we call → mergeSort → countPairs → merge
function added, rest all same as merge sort

⇒ r = mid + 1, cnt = 0
⇒ loop from i = lo to i <= mid
while → r <= hi & a[i] > (long long) 2 * a[r] → do r++;
cnt = cnt + (r − (mid + 1));
⇒ return cnt;

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

• Find 2 no.'s (in array) that sum = target. Same ele. twice is not allowed

⇒ Using unordered_map
⇒ Adding arr ele. & idx to mp
⇒ Doing a loop
  ↳ rem = target − num
  ↳ if (mp.find(rem) != mp.end() & mp[rem] != i)
    ↳ return {i, mp[rem]}
⇒ else return {-1, -1}

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

• Find 3 no.'s (in array) that, sum = target. No duplicate triplets in soln. set {i,j,k}

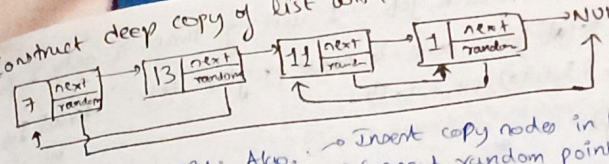
⇒ $T_c = O(n \log n) + O(n^2)$ $S_c = O(\text{total unique mp})$
⇒ We sort array, & now 1 pointer const & 2 pointers moving
⇒ loop from 0 to n
  ↳ if (i>0 & a[i]==a[i-1]) continue,
  ↳ j = i+1, k = n-1
  ↳ while (j<k)
    ↳ sum = add array i,j,k
    ↳ if(sum<0) → j++, else if (sum>0) → k--
    ↳ else
      ↳ add to 'ans' (2D vec) → a[i], a[j], a[k]
      ↳ do j++, k--
      ↳ loop(j<k & a[j]==a[j-1]) j++
      ↳ loop(j<k & a[k]==a[k+1]) k--
⇒ return ans;

• Find 4 no.'s (in array) such that sum = target ; i!=j!=k!=l ; No duplicate allowed in answer vector.

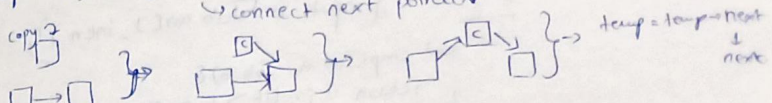
⇒ Here 2 pointers const. & 2 pointers moving
⇒ sort array
⇒ loop from 0 to n
  ↳ while (i>0 & a[i]==a[i-1]) continue
  ↳ for(j = i+1 → n)
    ↳ while (j!=i+1 & a[j]==a[j-1]) continue
    ↳ k = j+1, l = n-1
      ↳ while (k < l) (i,j,k,l)
        ↳ add all ele. to sum
        ↳ sum > tgt → l--, sum < tgt → k++
        ↳ add to 'ans' (2D) (a[i], a[j], a[k], a[l])
        ↳ k++, l--
        ↳ while (k<l & a[k]==a[k-1]) → k++
        ↳ while (k<l & a[l]==a[l+1]) → l--
⇒ return ans

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

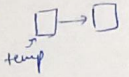
• Construct deep copy of list with random pointer



⇒ To create deep copy; Algo:
  → Insert copy nodes in between
  → connect random pointers
  ↳ connect next pointers

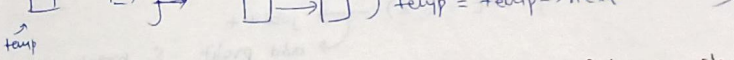


⇒ Inserting :



⇒ Connect random: while(temp! = NULL)
  ↳ copy = temp->next
  ↳ if(temp->random != NULL) copy->random = temp->random->next.
  temp = temp->next->next

⇒ Connect next i.e getDeepCopyList:
  ↳ {res->next = temp->next}
  ↳ {res = res->next}
  ↳ temp->next = temp->next->next
  ↳ temp = temp->next
  ↳ repeat



• Trapping rain water → width of bar = 1, given 'n' integers showing an elevation map. How much water can it trap after raining?

ans = 6

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

⇒ Start as left = 0 & right = n-1
⇒ While → left <= right
  ↳ if height of left is small/equal to right
    ↳ if → height of left more than leftMax → update 'leftMax'
    else → ans = maxLeft − height[left]
    left++
  ↳ else ↳ do above similar for right.
⇒ Since h[l] <= h[r] & curr. bar height is less than maxLeft, so we subtract & now we can say that it stores {subtract} value of water.

• Remove outermost parenthesis → ex: (()())(()) → ()()()

⇒ Maintain 'cnt' variable to know '(' & ')'.
⇒ loop string
  ↳ if → == '(' → if (cnt>0) → add to 'ans' string
    cnt++
  ↳ else → decrement 'cnt'
    ↳ if(cnt>0) → add to 'ans' string
⇒ return ans.

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

• Job-Sequencing Problem: each job → 1 unit of time & only 1 job at a time. Find total jobs done & max profit. Note: struct Job { int id; int deadline; int profit; }

ex: {{1,4,20}, {2,1,10}, {3,1,40}, {4,1,30}}

$T_c = O(n \log n) + O(N * M)$ (sorting) (N jobs) (M deadlines), $S_c = O(m)$

⇒ sort based on profit

{input is (Job arr[], int n)}

```
bool cmp(Job a, Job b){
    return a.profit > b.profit;
}
```

⇒ Find whose deadline is last → store in var. 'maxi'

⇒ Create an array of size 'maxi'.

⇒ Idea: we do job at it's deadline & if that day, we have already done some other job, then we must do it before deadline.

⇒ 2 for loops → 0→n

→ (arr[i].dead ; j>0 ; j--)

→ if slot = (-1) → jobs++ → add profit & break;

• Minimum no. of coins → deno.: [1, 2, 5, 10, 20, 50, 100, 200, 500, 2000]

Min. no. of coins needed to make change for Rs 'N'. {∞ supply}

⇒ We can use greedy as sum of previous any 2 coins is less than current coin value. ex: 20+10 = 30 < 50

⇒ loop → arr.size() - 1 → 0

→ while (n > arr[i])

→ n -= arr[i]

$T_c = O(N)$

$S_c = O(N)$

• Same as above question but denominations are not given in question. Find min. coins to make change for Rs 'N' {∞ supply}

ex: N = 30, M = 3, coins[] = {25, 10, 5}, ex: N = 11, M = 4, c[] = {9, 6, 5, 1}

⇒ Here you can see 9 < 6+5 → so it's a hint we can't use DP

⇒ Top-down $T_c = O(N*T)$, $S_c = O(N*T)$

→ We can see our target & index is only changing so create a DP

→ v<v<int>> dp(n, v<int>(target+1, -1))

• Recursive call → f(coins, n-1, target, dp)

→ if idx = 0 → if target % coins[idx] = 0 → return this division; → else 1e9

notTake → idx-1

take = INT_MAX, if (c[idx] <= target) → take → tgt - c[idx]

return dp[idx][tgt] = min(take, notTake);

⇒ Bottom-up

→ dp → initialised with '0'

→ target can be btw 0 ↔ target → so loop 0 → tgt

→ if t % coins[0] == 0 → dp[0][t] = t / coins[0]

→ else dp[0][t] = 1e9;

(2 loops)

→ ind → 1 to n

→ t = 0 → tgt

→ notTake = 0 + dp[i-1][t]

if(c[i] <= t) take = 1 + dp[i][t - c[i]];

dp[i][t] = min(take, notTake);

⇒ ans = dp[n-1][target] → if >= 1e9 → return -1

• Total subarrays having bitwise XOR of all ele. = 'B' in array 'A'.

⇒ ex: 1, 2, 3, 4, 5 — let xor = xR; let xor = x; let xor = B

$x \wedge B = xR$

$\Rightarrow x = xR \wedge B$

⇒ Means if we are at '4' & if we do 'xR^B' → if this ele. present → count++ → means a subarray is present till '4'

⇒ Declare unordered map, with mp[0]++ (0 = xr)

⇒ loop from 0 to end

→ xr = xr ^ a[i]

→ x = xr ^ b

→ cnt += mp[x]

→ mp[xr]++

$T_c = O(N)$

$S_c = O(N)$

• Longest substring without repeating characters → $T_c = O(N)$, $S_c = O(N)$

ex: pwwkew → op: 3; as → kew → only substring without repeating chars.

⇒ Using map & 2 pointer approach.

⇒ left = 0, ans = 0

⇒ loop string → (int right = 0)

→ if(mp.find(s[right]) != mp.end() && mp[s[right]] >= left)

→ left = mp[s[right]] + 1;

mp[s[right]] = right;

ans = max(ans, right - left + 1);

ex: 0 1 2 3 4 5 6 7 8 — c a b d z a b c d — l r

When r → 5 & l → 0, we see doing 'l++' will still have 'a' btw 'l' & 'r'. So we shift 'l' to → prev. r+1

• Min. no. of platforms needed at railway station, so that no train has to wait. Given arrival & departure times of trains. Same platform can't be used for arrival & departure of train at any instance of time.

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

⇒ Arrival, departure not associated with particular train, so we can sort both arrays

⇒ i=1, j=0

↳ while → i<n & j<n

↳ if(a[i] <= d[j]) plat++; i++;

else plat--; j--;

ans = max(ans, plat);

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

$S_c = O(1)$

• String compression → Append length of prefix made by a character just next to the char. (almost same, prev. also may be same)

ex: abcde → 1a1b1c1d1e

ex: aaaaaaaaaaaaaabb → 9a5a2b

⇒ We will have i/p as 'string → word'

⇒ while (idx(0) < n)

↳ char c = word[idx], cnt = 0

↳ while(idx < n & cnt < 9 & word[idx] == c) → idx++, cnt++

↳ comp += to_string(cnt) + c

$T_c = O(N)$

$S_c = O(N)$

⇒ return comp

• No. of substrings containing all 3 characters → string has 'a', 'b', 'c' only. Return no. of substrings containing at least 1 occurrence of above chars.

↳ ex: S = "abcabc"

↳ op: 10

⇒ vector<int> lastSeen(3, -1);

int cnt = 0;

for(i=0 to s.size()-1){

lastSeen[s[i] - 'a'] = i;

if(lastSeen[0] != -1 and lastSeen[1] != -1 and lastSeen[2] != -1){

cnt = cnt + 1 + min(lastSeen[0], min(lastSeen[1], lastSeen[2]));

}

}

return cnt;

• Max consecutive 1's → given binary array & 'k', return max no. of consecutive 1's if we can flip atmost 'k' 0's.

↳ ex: [1,1,1,0,0,0,1,1,1,1,0], k=2

↳ op: 6

⇒ len=0, cnt=0, i=0, j=0;

while(j < nums.size()){

if(nums[j] == 0) cnt++;

while(cnt > k){

if(nums[i] == 0) cnt--;

i++;

}

len = max(len, j-i+1);

j++;

}

• LC 1976 → Hint: Dijkstra's Algo → Find total no. of paths with min wt. from 0 to 'n-1'.

int countPaths(int n, vector<vector<int>>& roads){

vector<pair<int,int>>> adj(n);

vector<int> dist(n, INT_MAX), ways(n, 0);

for(auto it : roads){

adj[it[0]].push_back({it[1], it[2]});

adj[it[1]].push_back({it[0], it[2]});

}

priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

dist[0] = 0; ways[0] = 1; pq.push({0,0}); int mod = 1000000007;

while(!pq.empty()){

int dis = pq.top().first;

int node = pq.top().second;

pq.pop();

• Longest Increasing Subsequence → return length, (must be strictly ↑ing)

ex: [10,9,2,5,3,7,101,18] → op: 4 → [2,3,7,101]

⇒ int lengthOfLIS(vector<int> arr){

int n = arr.size();

vector<vector<int>> t(n+1, vector<int>(n+1, 0));

for(int idx = 0; idx < n; idx++){

for(int prev_idx = 0; prev_idx <= idx; prev_idx++){

t[idx+1][prev_idx] = max(t[idx+1][prev_idx], t[idx][prev_idx]);

if(prev_idx == 0 or arr[idx] > arr[prev_idx - 1]){

t[idx+1][idx+1] = max(t[idx+1][idx+1], 1 + t[idx][prev_idx]);

}

}

}

int result = 0;

for(int i = 1; i <= n; i++) result = max(result, t[n][i]);

return result;

}

⇒ vector<int> temp; temp.push_back(nums[0]);

for(int i = 1; i < n; i++){

if(temp.back() < nums[i]) temp.push_back(nums[i]);

else{

int idx = lower_bound(temp.begin(), temp.end(), nums[i]) - temp.begin();

temp[idx] = nums[i];

}

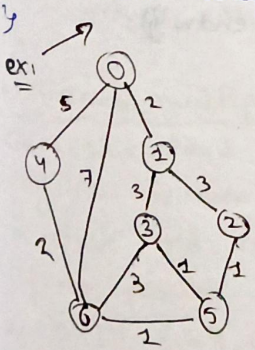
}

return temp.size();

• LC 1976 → Hint: Dijkstra's Algo → Find total no. of paths with min. wt. from 0 to 'n-1'.

```
int countPaths(int n, vector<vector<int>>& roads){
    vector<list<pair<int,int>>> adj(n);
    for(auto it: roads){
        adj[it[0]].push_back({it[1], it[2]});
        adj[it[1]].push_back({it[0], it[2]});
    }
    priority_queue<pair<int,int>, vector<pair<int,int>>,
                                  greater<pair<int,int>>> pq;

    vector<int> dist(n, INT_MAX), ways(n,0);
    dist[0] = 0; ways[0] = 1; pq.push({0,0}); int mod = 10000007;
    while(!pq.empty()){
        int dis = pq.top().first;
        int node = pq.top().second;
        pq.pop();
        for(auto it: adj[node]){
            int adjNode = it.first;
            int edW = it.second;
            if(dis+edW < dist[adjNode]){   // first time coming with this short distance
                dist[adjNode] = (dis+edW)%mod;
                pq.push({(dis+edW)%mod, adjNode});
                ways[adjNode] = ways[node];
            }
            else if((dis+edW) == dist[adjNode]){   // if we again encounter a node with same short distance, we increment the no. of ways.
                ways[adjNode] = (ways[adjNode] + ways[node])%mod;
            }
        }
    }
    return ways[n-1];
}
```

ex:

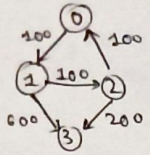


n = 7, roads = [[0,6,7], [0,1,2], [1,2,3], [1,3,3], [6,3,3], [3,5,1], [6,5,1], [2,5,1], [0,4,5], [4,6,2]]

o/p: 4

• LC 787

Ex:



n = 4, flights = [[0,1,100], [1,2,100], [2,0,100], [1,3,600], [2,3,200]], src = 0, dst = 3, k = 1

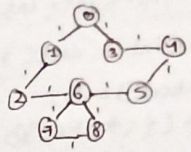
o/p: 700

Given 'src', 'dst', 'k', return cheapest way from 'src' to 'dst' with atmost 'k' stops.

```
int cheapestPrice (int n, vector<vector<int>> flights, int src, int dest, int k){
    vector< list< pair <int,int>>> graph(n);
    for(int i=0 ; i< flights.size() ; i++) graph[flights[i][0]].push_back
                                        ({flights[i][1], flights[i][2]});
    vector <int> dist(n, INT_MAX);
    queue <pair< int, pair <int, int>>>q; //{stops, {node, dist}}
    q.push ({0, {src, 0}});
    dist[src] = 0;
    while(!q.empty()){
        auto curr = q.front();
        q.pop();
        int stops = curr.first, node = curr.second.first, price = curr.second
                                                                .second;
        if(stops > k) continue;
        for(auto it : graph[node]){
            int adjNode = it.first;
            int endW = it.second;
            if(price+endW < dist[adjNode] and stops <= k){
                dist[adjNode] = price + endW;
                q.push ({stops+1, {adjNode, price+endW}});
            }
        }
    }
    if(dist[dst] == INT_MAX) return -1;
    return dist[dst];
}
```

• Shortest Path in Undirected Graph

unit wt edges, find shortest path from 'src' to all the vertex. If any vertex = unreachable → return -1 for that vertex



ex: n = 9, m = 10, src = 0,
edges = [[0,1],[0,3],[3,4],[4,5],[5,6],[1,2],[2,6],[6,7],[7,8],[6,8]]

o/p: 0 1 2 1 2 3 3 4 4

```
vector <int> shortestPath (vector<vector<int>>edges, int v, int m, int src){
    vector<vector <int>> adj(v);
    vector <int> dist(v, INT_MAX);
    for(auto it : edges){
        adj[it[0]].push_back(it[1]);
        adj[it[1]].push_back(it[0]);
    }
    dist[src] = 0;
    queue <int> q;
    q.push(src);
    while(!q.empty()){
        int node = q.front();
        q.pop();
        for(auto it : adj[node]){
            if(dist[it] > 1+ dist[node]){
                dist[it] = 1+dist[node];
                q.push(it);
            }
        }
    }
    vector<int> ans(v, -1);
    for(int i=0; i<v; i++){
        if(dist[i] != INT_MAX) ans[i] = dist[i];
    }
    return ans;
}
```

$T_C = O(N+E)$

$S_C = O(N)$

• Minimum multiplications to reach End

Given start, end & array = arr. At each step, start multiplied with any no. & then mod with 10^5 to get new start. Find min. steps in which end can be achieved starting from 'start'. Return -1, if not possible to reach end.

ex: arr[] = {3, 4, 65}, start = 7, end = 66175 → op: 4

S1 → 7×3 = 21

S2 → 21×3 = 63

S3 → 63×65 = 4095

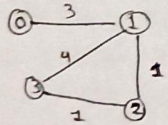
S4 → 4095×65 = 266175 % 100000 = 66175 ✓

```
int minimumMultiplication(vector<int>& arr, int start, int end){
    int mod = 100000, n = arr.size();
    queue<pair<int,int>> q;
    vector<int> dist(100000, INT_MAX);
    dist[start] = 0;
    q.push({start, 0});
    if(start == end) return 0;
    while(!q.empty()){
        int node = q.front().first;
        int steps = q.front().second;
        q.pop();
        for(int i=0; i<arr.size(); i++){
            int num = (arr[i] * node) % mod;
            if(dist[num] > steps+1){
                dist[num] = steps+1;
                if(num == end) return steps+1;
                q.push({num, steps+1});
            }
        }
    }
    return -1;
}
```

• LC 1334

Return the city with smallest no. of cities that are reachable & whose distance is atmost 'distanceThreshold'. If there are multiple such cities, return the city with the greatest no.

eg:



n=4, edges = [[0,1,3], [1,2,1], [1,3,4], [2,3,1]]
distanceThreshold = 4

op: 3

explanation:
City 0 → [1, 2]
City 1 → [0, 2, 3]
City 2 → [0, 1, 3]
City 3 → [1, 2]

∴ Ans = 0, 3

Since 3 > 0 ⇝ ∴ answer = 3

```
//Hint → Floyd's Warshall Algo.:
int findTheCity(int n, vector<vector<int>>& edges, int distanceThreshold)
{
    vector<vector<int>> dist(n, vector<int>(n, INT_MAX));
    for(auto it : edges){
        dist[it[0]][it[1]] = it[2];
        dist[it[1]][it[0]] = it[2];
    }
    for(int i=0; i<n; i++) dist[i][i] = 0;
    for(int k=0; k<n; k++){
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                if(dist[i][k]==INT_MAX or dist[k][j]==INT_MAX){
                    continue;
                }
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }
        }
    }
    int cityNo = -1, cntCity = n;
    for(int city = 0; city<n; city++){
        int cnt = 0;
        for(int adjCity = 0; adjCity<n; adjCity++){
            if(dist[city][adjCity] <= distanceThreshold) cnt++;
        }
        if(cnt <= cntCity){
            cntCity = cnt;
            cityNo = city;
        }
    }
    return cityNo;
}
```

• LC 214

Return shortest palindrome by adding characters in-front of 's'?

eg: s = "aacecaaa" ⇝ op: "aaacecaaa"

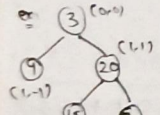
Brute force → reverse whole string & add it to beginning of 's'

↳ optimized → KMP Algorithm

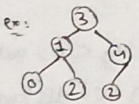
```
string shortestPalindrome(string s){
    int n = s.size(), i=0, j=1;
    string rs = s;
    reverse(rs.begin(), rs.end());
    rs = s + '#' + rs;
    int n2 = rs.size();
    vector<int> lps(n2, 0);
    while(j<n2){
        if(rs[i] == rs[j]){
            lps[j] = i+1;
            i++; j++;
        }
        else{
            if(i==0){
                lps[j]=0;
                j++;
            }
            else i=lps[i-1];
        }
    }
    string temp = "";
    for(int i=lps[n2-1]; i<n; i++){
        temp += s[i];
    }
    reverse(temp.begin(), temp.end());
    string res = temp+s;
    return res;
}
```

• LC 987 → $T_c = O(N \log N)$

Calculate vertical order traversal of b-tree.
↳ top-bottom & left-right ordering.

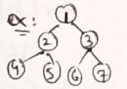
ex: 

op: [[9],[3,15],[20],[7]]

ex: 

op: [[0],[1],[3,2,2],[4]]

There may be multiple nodes in same row & same column. In such a case, sort these nodes by value.

ex: 

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

```
vector<vector<int>> verticalTraversal(Node* root){
    if(root == NULL) return {{}};
    queue<pair<Node*, pair<int,int>>> q; //pair of node & its (column, row)
    q.push({root, {0,0}});
    map<int, vector<pair<int,int>>> mp; //map of column to pairs of (row, value)
    while(!q.empty()){
        int nodesAtCurrentLevel = q.size();
        while(nodesAtCurrentLevel--){
            auto node = q.front();
            q.pop();
            Node* currNode = node.first;
            int col = node.second.first;
            int row = node.second.second;
            mp[col].push_back({row, currNode->val});
            if(currNode->left != NULL) q.push({currNode->left, {col-1, row+1}});
            if(currNode->right != NULL) q.push({currNode->right, {col+1, row+1}});
        }
    }
    vector<vector<int>> ans;
    for(auto x : mp){
        vector<int> temp;
        sort(x.second.begin(), x.second.end()); //Sort based on row first, then value.
        for(auto z : x.second){
            temp.push_back(z.second);
        }
        ans.push_back(temp);
    }
    return ans;
}
```

• Count Palindromic Subsequences

Find no. of palindromic subsequence (need not be distinct) present in 'str'.
Return answer % 10^9+7.

ex: abca → op: 7 → op: a, b, c, a, aa, aba, aca

ex: abcd → op: 4 → op: a, b, c, d

```
int mod = 1000000007;
long long int solve(string &s, int i, int j, vector<vector<int>> &t){
    if(i>j) return 0;
    if(i==j) return t[i][j] = 1; // a single character is a palindrome
    if(t[i][j] != -1) return t[i][j];
    int result = 0;
    if(s[i] == s[j]){
        result = (1 + solve(s,i+1,j,t) + solve(s,i,j-1,t)) % mod;
    }
    else{
        result = (mod + solve(s,i+1,j,t) + solve(s,i,j-1,t)
                      - solve(s,i+1,j-1,t)) % mod;
    }
    return t[i][j] = result;
}

long long int countPS(string s){
    int n = s.size();
    vector<vector<int>> t(n, vector<int>(n,-1));
    return solve(s,0,n-1,t);
}
```

//MOD PROPERTY = $(a-b) \% m = (m + a \% m + b \% m) \% m$

Explanation: "abca"

abca → 7

s[i] == s[j] → Add 1 + (_) + (_)

1 + bca + abc

bc + ca - c (3)

ab + bc - b

Here in bc, ca; 'c' is getting considered 2 times, so we subtract it.

Here 'c' → {i+1, j-1} chars

- **LC 1514** : find max. prob. of path from 'start' to 'end'. (0 ≤ ... ≤ 1)

* Similar to LC 1976 (written previously)

↳ ex: 0.5, 0.2, 0.5 ; s=0, e=2 ; o/p: 0.25

```
double maxProbability(int n, vector<vector<int>>& edges, vector<int> succProb,
                      int start, int end){

    vector<list<pair<int,int>>> graph(n);
    for(int i=0; i<edges.size(); i++){
        graph[edges[i][0]].push_back({edges[i][1], succProb[i]});
        graph[edges[i][1]].push_back({edges[i][0], succProb[i]});
    }
    priority_queue<pair<double,int>> pq; // max heap
    vector<double> maxProb(n, 0.0);
    maxProb[start]=1.0;
    pq.push({1.0, start});          // iterating through graph with help of priority queue
    while(!pq.empty()){
        double prob = pq.top().first;
        int node = pq.top().second;
        pq.pop();
        if(node==end) return prob;
        for(auto it: graph[node]){          // if we found a path with higher prob., update & push it to pq.
            int adjNode = it.first;
            double edgeProb = it.second;
            if(prob*edgeProb > maxProb[adjNode]){
                maxProb[adjNode] = prob*edgeProb;
                pq.push({maxProb[adjNode], adjNode});
            }
        }
    }
    return 0.0;     // if we never reach end node
}
```

- **Inorder Successor in BST** ; find inorder successor of Node 'x'

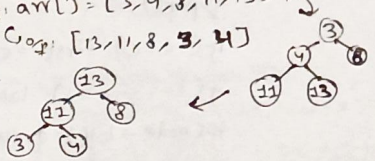
ex: (20, 8, 22, 4, 12, 10, 14) , K (data of x) = 8 → o/p: 10

```
Node* inorderSuccessor(Node* root, Node* x){
    Node* successor = NULL;
    while(root!=NULL){
        if(root->data > x->data){
            successor = root;          // potential answer → so store
            root = root->left;
        }
        else root = root->right;
    }
    return successor;
}
```

$T_c = O(H)$

$S_c = O(1)$

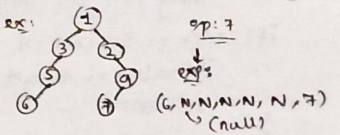
- **Convert min Heap to max Heap** ; ex: arr[] = [3, 4, 8, 11, 13] → o/p: [13, 11, 8, 3, 4]



```
void heapify(vector<int>& arr, int n, int i){
    int largest = i;
    int left = 2*i+1;
    int right = 2*i+2;
    if(left<n and arr[left] > arr[largest]) largest = left;
    if(right<n and arr[right] > arr[largest]) largest = right;
    if(largest != i){
        swap(arr[i], arr[largest]); // Swap root with largest
        heapify(arr, n, largest); // Recursively heapify the affected sub-tree
    }
}
void convertMinToMaxHeap(vector<int>& arr, int n){
    for(int i=(n/2-1); i>=0; i--) heapify(arr, n, i);
}
```

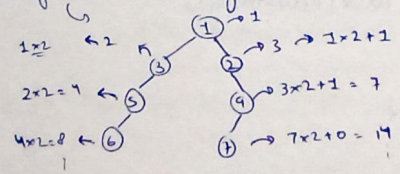
↳ Start from last non-leaf node & heapify each node in reverse level order.

- **LC 662** : find max width of binary tree ; ex: o/p: 7



(6, N, N, N, N, 7) (null)

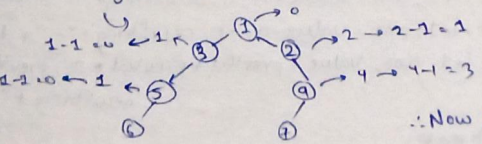
Brute force → mark every node {1-based idx}



But since no. ↑, so might cause overflow error.

Optimal → $T_c = O(N)$, $S_c = O(N)$

↳ Subtract min. from each level {0-based idx}



∴ Now no.s are btw [0, n) (so no overflow)

1 based idx → child = 2×i, 2×i+1

0 based idx → child = 2×i+1, 2×i+2

```
int widthOfBinaryTree(Node* root){
    if(root == NULL) return 0;
    queue<pair<Node*, int>>q;
    int ans = 0;
    q.push({root,0});
    while(!q.empty()){
        int size = q.size();
        int firstMin = q.front().second;
        int first, last;
        for(int i=0; i<size; i++){
            int curr_id = q.front().second - firstMin;
            Node* node = q.front().first;
            q.pop();
            if(i==0) first = curr_id;
            if(i==size-1) last = curr_id;
            if(node->left) q.push({node->left, curr_id*2+1});
            if(node->right) q.push({node->right, curr_id*2+2});
        }
        ans = max(ans, last-first+1);
    }
    return ans;
}
```

- Refer LC 1905 → Similar to "No. of Islands" → Book 2

- **LC 282** : Given string 'nums' → only digits & int. 'target'. Return all possibilities to insert: '+', '-', '*' btw digits of nums so that resultant value = target. (See examples next pages)

```
void solve(string& s, int& target, long index, vector<string>& ans, long value,
           long prevVal, string newString){
    if(index == s.size()){
        if(value == target) ans.push_back(newString);
        return;
    }
    string temp = ""; long n = 0;
    for(long i=index; i<s.size(); i++){
        temp += s[i];
        n = n*10 + (s[i]-'0');
        if(index==0) solve(s, target, i+1, ans, n, n, temp);
        else{
            solve(s, target, i+1, ans, value+n, n, newString+"+"+temp);
            solve(s, target, i+1, ans, value-n, -n, newString+"-"+temp);
            solve(s, target, i+1, ans, value-prevVal+prevVal*n, prevVal*n,
                  newString+"*"+temp);
        }
        if(s[index]=='0') break;
    }
}
```

```
vector<string> addOperators(string s, int target){
    vector<string> ans;
    solve(s, target, 0, ans, 0, 0, "");
    return ans;
}
```

(Examples)

1) num = "123", target = 6
op: ["1*2*3", "1+2+3"]

2) num = "7024", target = 9
op: ["7+0-2+4", "7-0-2+4"]

3) num = "58824", target = 6
op: []

- **LC 23**: Merge all linked lists into 1 sorted linked list & return it.

ex: lists = [[1,4,5],[1,3,4],[2,6]] op: [1,1,2,3,4,4,5,6]

```
Node* mergeKLists(vector<Node*>& lists){
    priority_queue<pair<int,Node*>, vector<pair<int,Node*>>,
                   greater<pair<int,Node*>>> pq;
    Node* dummy = new Node(-1);
    Node* temp = dummy;
    for(int i=0; i<lists.size(); i++){
        if(lists[i]!=nullptr) pq.push({lists[i]->val, lists[i]});
    }
    while(!pq.empty()){
        pair<int,Node*> p = pq.top();
        pq.pop();
        temp->next = p.second;
        temp = temp->next;
        if(p.second->next != nullptr){
            pq.push({p.second->next->val, p.second->next});
        }
    }
    return dummy->next;
}
```

(K×logK) — points to the pointer of header of lists[i] → That's why we are able to access 1st val. of each lists[i]

$T_c = O(\log k)$ → k = size of lists

runs for K×N (size of 1 list) (total lists) — log(k)

$\therefore T_c : K \times \log K + N \times K \times \log K$; $S.S : O(K)$

- **LC 66** : Plus One to the digits

ex: digits = [1,2,3] → op: [1,2,4] & digits = [9] → op: [1,0]

```
vector<int> plusOne(vector<int>& digits){
    vector<int> ans = digits;
    int n = ans.size(), carry = 1;
    for(int i=n-1; i>=0; i--){
        int num = ans[i] + carry;
        ans[i] = num%10;
        carry = num/10;
    }
    if(carry>0){
        ans.push_back(0);
        n = ans.size();
        for(int i=n-1; i>0; i--){
            ans[i] = ans[i-1];
        }
        ans[0] = carry;
    }
    return ans;
}
```

LC 41 : First Missing Positive

(return smallest (+)ve integer that's not present in nums) → $T_c = O(n)$, $S_c = O(1)$

ex: nums : [3,4,-1,1] op: 2 , nums : [6,1,5,2,4,3] op: 7 , nums: [7,8,9,11,12] op: 1

```
int firstMissingPositive(vector<int>& nums){
    int n=nums.size();
    for(int i=0; i<n; i++){
        while(nums[i]>=1 and nums[i]<=n and nums[nums[i]-1]
                                        != nums[i]){
            swap(nums[i], nums[nums[i]-1]);
        }
    }
    for(int i=0; i<n; i++){
        if(nums[i] != i+1) return i+1;
    }
    return n+1;
}
```

LC 241 : Different ways to add Parentheses

ex: expression = "2*3-4*5" op: [-34, -14, -10, -10, 10]

explanation: (2*(3-(4*5))) = (-34)
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10

expression has → range = [0,99]
↳ '+', '-', '*'

```
vector<int> diffWaysToCompute(string expression){
    vector<int> ans;
    for(int i=0; i<expression.size(); i++){
        char c=expression[i];
        if(c=='*' or c=='-' or c=='+'){
            vector<int> left, right;
            left = diffWaysToCompute(expression.substr(0,i));
            right = diffWaysToCompute(expression.substr(i+1));
            for(auto i: left){
                for(auto j: right){
                    if(c=='*') ans.push_back(i*j);
                    else if(c=='+') ans.push_back(i+j);
                    else ans.push_back(i-j);
                }
            }
        }
    }
    if(ans.empty()) ans.push_back(stoi(expression));
    return ans;
}
```

ex: 2*3-4*5 (-11, -5) → so for(-){ for(-){ } } checks all the comb. on the left & right side of 'operator'

LC 725 : Split Linked List in Parts

Given 'k' → split 'LL' into 'k' parts. Length of parts must be equal and cannot differ in size by more than one, if any 2 parts compared.

ex: 1→2→3 , k=5 op: [[1],[2],[3],[],[]]

ex: 1→2→3→4→5→6→7 , k=2 op: [[1,2,3,4],[5,6,7]]

```
vector<Node*> splitListToParts(Node* head, int k){
    vector<Node*> ans(k, NULL);
    int size = 0;
    Node* temp = head;
    while(temp != NULL){
        temp = temp->next;
        size++;
    }
    int len = size/k, extra = size%k;
    temp = head;
    for(int i=0; i<k; i++){
        ans[i] = temp;
        int partLength = len + (extra>0 ? 1 : 0);
        extra--;
        for(int j=1; j<partLength; j++){
            if(temp!=NULL) temp = temp->next;
        }
        if(temp!=NULL){
            Node* nextPart = temp->next;
            temp->next = NULL;
            temp = nextPart;
        }
    }
    return ans;
}
```

Shortest Prime Path : convert num1 to num2 by altering only a single digit at a time, such that every num. that we get is a prime no.

$num = 4 digit always$

ex: num1 = 1033, num2 = 8179 → op: 6

expl: 1033 → 1733 → 3733 → 3779 → 8779 → 8179

```
int solve(int num1, int num2){
    if(num1 == num2) return 1;
    vector<int> prime(10000, 1);
    for(int i = 2; i<10000; i++){
        if(prime[i]) for(int j=2*i; j<10000; j+=i) prime[j]=0;
    }
    unordered_set<int> st;
    for(int i=0; i<10000; i++) if(prime[i]) st.insert(i);
    queue<pair<int,int>> q;
    q.push({0, num1});
    st.erase(num1);  // so that that prime no. isn't repeated.
    while(!q.empty()){
        int dist = q.front().first, num = q.front().second;
        q.pop();
        if(num == num2) return dist;
        string number = to_string(num);
        for(int i=0; i<4; i++){
            for(int j=0; j<=9; j++){
                char c = number[i];
                number[i] = '0'+j;
                int newNum = stoi(number);
                if(newNum >= 1000 and st.find(newNum) != st.end()){
                    st.erase(newNum);
                    if(newNum == num2) return dist+1;
                    q.push({dist+1, newNum});
                }
                number[i] = c;
            }
        }
    }
    return -1;
}
```

- **LC 316** : Remove Duplicate Letters

Given string 's' → remove duplicate letters so that every letter appears once & only once. Result = smallest in lexographical order among all possible results.

ex: "cdadabcc" → "adbc" , "ecbacba" → "eacb"

A) Intuition: → Add ele. in stack, if any ele. lexographically smaller is next, then check if st.top() is available after the next ele. in the string → if yes → remove st.top() else → add next ele.

```
string removeDuplicateLetters(string s){
    vector<int> lastIndex(26, 0);
    for(int i=0; i<s.size(); i++) lastIndex[s[i]-'a'] = i;
    vector<char> seen(26, false);
    stack<char> st;
    for(int i=0; i<s.size(); i++){
        int curr = s[i]-'a';
        if(seen[curr]) continue;
        while(st.size()>0 and st.top()>s[i] and i<lastIndex[st.top()-'a']){
            seen[st.top()-'a'] = false;
            st.pop();
        }
        st.push(s[i]);
        seen[curr] = true;
    }
    string ans = "";
    while(st.size()>0){
        char x = st.top();
        st.pop();
        ans+=x;
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

- **LC 1043** : divide 'arr' into subarrays of length atmost 'k'. After partioning, each subarray has their val. changed to max. val. of that subarray. Return largest sum of given array after partitioning.

ex: [1,15,7,9,2,5,10], k=3 → o/p: 84 → ex: [1,15,7][9][2,5,10] → 15+15+15+9+10+10+10

```
int maxSum(vector<int>&arr, int k){
    int n=arr.size();
    vector<int> t(n+1,0);
    for(int i=n-1; i>=0; i--){
        int maxVal=0, ans=0;
        for(int j=i; j<min(i+k,n); j++){
            maxVal = max(maxVal, arr[j]);
            ans = max(ans, (j-i+1)*maxVal + t[j+1]);
        }
        t[i] = ans;
    }
    return t[0];
}
```

- **LC 312** : Given array 'nums' = balloons. If i^{th} balloon is burst, we will get nums[i-1] * nums[i] * nums[i+1] coins. If i-1, i+1 = out of bounds, treat it as 1. Return max. coins you can collect by bursting the balloons wisely.

ex: [3,1,5,8] → exp: [3,1,5,8] → [3,5,8] → [3,8] → [8] → []

Op: 167

A) Top-down $T_c = N \times N \times N$ ($O(N^3)$), $S_c = O(N^2)$

```
int maxCoinsHelper(int i, int j, vector<int>&nums, vector<vector<int>>& dp){
    if(i>j) return 0;
    if(dp[i][j]!=0) return dp[i][j];
    int maxCoins = INT_MIN;
    for(int k=i; k<=j; k++){
        int coins = nums[i-1] * nums[k] * nums[j+1];   // Last balloon which is burst
        int remainingCoins = maxCoinsHelper(i, k-1, nums, dp) +
                             maxCoinsHelper(k+1, j, nums, dp);
        maxCoins = max(maxCoins, coins + remainingCoins);
    }
    return dp[i][j] = maxCoins;
}
int maxCoins(vector<int>&nums){
    int n=nums.size();
    nums.insert(nums.begin(),1); nums.push_back(1);
    vector<vector<int>> dp(n+2, vector<int>(n+2,-1));
    return maxCoinsHelper(1,n,nums,dp);
}
```

Bottom-up:

```
int maxCoins(vector<int>&nums){
    // I
    for(int i=n; i>=1; i--){
        for(int j=1; j<=n; j++){
            if(i>j) continue;
            int maxi = INT_MIN;
            for(int ind=i; ind<=j; ind++){
                int cost = nums[i-1]*nums[ind]*nums[j+1] + t[i][ind-1]
                           + t[ind+1][j];
                maxi = max(maxi, cost);
            }
            t[i][j] = maxi;
        }
    }
    return t[1][n];
}
```

balloons bursted from 'i' to 'j'

• LC399: ip: equations = [["a","b"],["b","c"]], values = [2.0, 3.0].

($\frac{A_i}{B_i}$ = values[i]) (A_i, B_i)

queries = [["a","c"],["b","a"],["a","e"],["a","a"],["x","x"]].

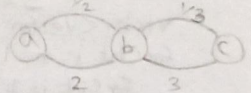
Find o/p for queries? → (in this case o/p: [6.0, 0.5, -1.0, 1.0, -1.0])

If answer can't be determined, return -1!

A)

```
void dfs(unordered_map<string, vector<pair<string, double>>>& adj,
    string src, string dst, unordered_set<string>& visited, double product
    , double & ans){
    if(visited.find(src) != visited.end()) return;
    visited.insert(src);
    if (src == dst){
        ans = product;
        return;
    }
    for(auto p: adj[src]){
        string v = p.first;
        double val = p.second;
        dfs(adj, v, dst, visited, product*val, ans);
    }
}
vector<double> calcEquation(vector<vector<string>>& equations,
    vector<double>& values, vector<vector<string>>& queries){
    int n = equations.size();
    unordered_map<string, vector<pair<string, double>>> adj;
    for(int i=0; i<n; i++){
        string u = equations[i][0], v = equations[i][1];
        double val = values[i];
        adj[u].push_back({v, val});
        adj[v].push_back({u, 1.0/val});
    }
    vector<double> result;
    for(auto query: queries){
        string src = query[0], dst = query[1];
        double ans = -1.0, product = 1.0;
        if(adj.find(src) != adj.end()){
            unordered_set<string> visited;
            dfs(adj, src, dst, visited, product, ans);
        }
        result.push_back(ans);
    }
    return result;
}
```

Graph for above:

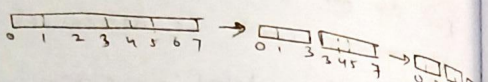


$T_c = O(N(V+E))$

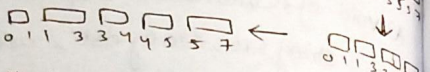
• LC 1547 : Min. Cost to Cut a Stick.

Wooden Stick length = n . Given array 'cuts' where cuts[i] denotes a position you should perform a cut. Cost of 1 cut = length of Stick to be cut. Total cost = sum of costs of all cuts.
Return min. total cost of cuts

ex: n=7, cuts = [1,3,4,5] →



Optimal order = [3,5,1,4]
↳ Performing cuts in these order
↳ : cost = 7+4+3+2 = 16



A) (Top-down) → $T_c : O(N^3)$, $S_c = O(N^2)$

```
int dfs(int i, int j, vector<int>& cuts, vector<vector<int>>& t){
    if(i>j) return 0;
    if(t[i][j] != -1) return t[i][j];
    int mini = INT_MAX;
    for(int k = i; k<=j; k++){
        int cost = cuts[j+1] - cuts[i-1] + f(i, k-1, cuts, t) + f(k+1, j, cuts, t);
        mini = min(mini, cost);
    }
    return t[i][j] = mini;
}
int minCost(int n, vector<int>& cuts){
    int m = cuts.size();
    cuts.push_back(n); cuts.insert(cuts.begin(), 0);
    sort(cuts.begin(), cuts.end());
    vector<vector<int>> t(m+1, vector<int>(m+1, -1));
    return f(1, m, cuts, t);
}
```

(we are calling i→1 to n & j→n to 1. So in tabulation, we reverse it. So we run the loop of i→n to 1 & j→1 to n)

(Bottom-up)

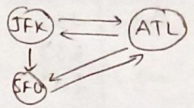
```
int minCost(int n, vector<int>& cuts){
    vector<vector<int>> t(m+2, vector<int>(m+2, 0));
    for(int i = m; i>=1; i--){
        for(int j = 1; j<=m; j++){
            if(i>j) continue;
            int mini = INT_MAX;
            for(int k = i; k<=j; k++){
                int cost = cuts[j+1] - cuts[i-1] + t[i][k-1] + t[k+1][j];
                mini = min(mini, cost);
            }
            t[i][j] = mini;
        }
    }
    return t[1][m];
}
```

• LC 332 : Reconstruct Itinerary

tickets[i] = [from_i, to_i] representing departure & arrival airports, (airline tickets) reconstruct the itinerary starting from "JFK".
If multiple valid itineraries exist, return the one with the smallest lexical order. Use all tickets exactly once.

ex: tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]

⇒ ["JFK","ATL","JFK","SFO","ATL","SFO"]



a) unordered_map<string, priority_queue<string, vector<string>, greater<string>>> adj;

```
vector<string> result;
void dfs(string u){
    priority_queue<string, vector<string>, greater<string>>& edges = adj[u];
    while(!edges.empty()){
        string v = edges.top();
        edges.pop();
        dfs(v);
    }
    result.push_back(u);
}
vector<string> findItinerary(vector<vector<string>>& tickets){
    for(auto e : tickets){
        adj[e[0]].push(e[1]);
    }
    dfs("JFK");
    reverse(result.begin(), result.end());
    return result;
}
```

• LC 2035 : Partition Array into 2 arrays (size 2*n, size n) to minimize sum difference between both the arrays. Return the min. sum.

ex: nums = [2,-1,0,4,-2,-9] ⇒ ans: 0 ↝ exp. ((2+4+(-9)) - (-1+0-2)) = 0

A) Prerequisites: 1) Meet in the middle algo.

2) Power Set (using bitmasking)

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

Meet in the middle algo.:

ex: Given set of int. $N (\le 40)$ & each val. $\le 10^7$. Determine max. sum subset having sum $\le S$, where $S \le 10^8$

eg: a[] = [5, 3, -1, 7] , S = 13

Soln. 1 : All possible subsets $\rightarrow 2^n$:

Size 0	Size 1	Size 2		Size 3	Size 4
{}	{5}	{5,3} = 8	{3,-1} = 2	{5,3,-1} = 7	{5,3,-1,7} = 14
	{3}	{5,-1} = 4	{3,7} = 10	{5,3,7} = 15	
	{-1}	{5,7} = 12	{-1,7} = 6	{5,-1,7} = 11	
	{7}			{3,-1,7} = 9	

$T_c = O(2^n)$

$\therefore$ Ans = 12 (nearest to 13)

Soln 2: [5, 3, -1, 7]

[5, 3] [-1, 7]

[0, 5, 3, 8] [0, -1, 7, 6]

(Sort)

[-1, 0, 6, 7]

x y

$\therefore$ **Step 1:** Divide array into 2 equal parts

Step 2: Find all possible subset sums

Step 3: Sort the 2nd array

Step 4: $x + y \le Sum$

$\hookrightarrow \therefore y \le (sum - x)$

(Using binary search for this)

$\therefore$ **Tc Analysis:**

- All subset sums = $2^{N/2} + 2^{N/2}$
- Sort 2nd array = $2^{N/2} \log_2 2^{N/2} = \frac{N}{2} \cdot 2^{N/2}$
- Binary Search on 2nd array = $\log_2 2^{N/2} = N/2$

  No. of times binary search is triggered = size of 1st array = $2^{N/2}$

  $\therefore$ Total search time = $\frac{N}{2} \cdot 2^{N/2}$

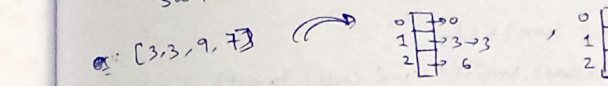
$\therefore$ Total $T_c = 2^{N/2} + 2^{N/2} + \frac{N}{2} \cdot 2^{N/2} + \frac{N}{2} \cdot 2^{N/2} = O(N \cdot 2^{N/2})$

$\therefore$ Now in current prob. $\rightarrow$ we will generate all sum possible using bitmasking & then use meet in the middle algo.

$\therefore$ **Step 1:** Partition array into 2 parts. ex: [3, 3, 9, 7] $\rightarrow$ [3, 3] [9, 7] $\}$ $T_c = O(1)$

Step 2: Find all possible sums of both arrays & organize them according to how many ele.s were included to form the sum.

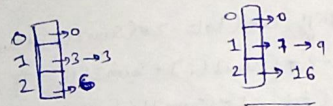
ex: [3, 3, 9, 7]



row = no. of items included

col = sum values possible

Step 3: Find the min. diff. (take 'k' items from a1[] & (n-k) items from a2[])



(Sort the 2nd list (MIM Algo.))

min. Diff = ∞

Sum = 3+3+9+7 = 22

Target = 11

- x = 0 $\rightarrow$ 11 - 0 = 11 $\rightarrow$ lowerbound = 16 $\rightarrow$ $\therefore$ 16 - 11 = 5 $\rightarrow$ 2*5 = 10 $\rightarrow$ $\therefore$ min. Diff = 10

  (we got 5) (multiplied '2' to get the distance) (equally far)

- x = 1 $\rightarrow$ 11 - 3 = 8 $\rightarrow$ lb = 9

  abs(11 - 11) = (11) $\rightarrow$ x2 = 2 (<10) $\rightarrow$ $\therefore$ min. Diff = 2 -- (& continue)

  (9+3)

We are aiming to minimize the abs. diff. btw the sum of 2 partitions & ideally we want each subarray to sum upto, target

$$target = \frac{total\ sum}{2}$$

$T_c = O(1 + N \cdot 2^n + N \cdot 2^n + 2^n \log 2^N)$

(partition array) (find all possible sums) (for each item in a1[], apply BS in a2[] list.)

- In this que. $\rightarrow$ array ele.s = (+) & (-)

In Aditya Verma DP $\rightarrow$ I) 4) Minimize subset sum difference

$\downarrow$

(Only (+)ve values included)

(so we can't use it here.)

Code:

```
int Solve(vector<int>& sum1, vector<int>& sum2, int tot){
    int m = sum1.size(), n=sum2.size();
    int minDiff = INT_MAX;
    for(int i = 0; i<m; i++){
        int target = tot/2 - sum1[i];
        int lb = lower_bound(sum2.begin(),sum2.end(),target) - sum2.begin();
        if(lb == sum2.size()) minDiff = min(minDiff, abs(tot - 2*(sum1[i] +
                                                    sum2[lb-1])));
        else if(lb==0) minDiff = min(minDiff, abs(tot -2*(sum1[i] + sum2[lb])));
        else minDiff = min(minDiff, abs(tot - 2*(sum1[i] + sum2[lb-1])),
                                    abs(tot - 2*(sum1[i] + sum2[lb])));
    } (sum2[lb-1] < target ≤ sum2[lb])
    return minDiff;
}
int minimizeDifference(vector<int>& nums){
    int n = nums.size()/2, tot=0;
    vector<int> left, right;
    for(int i=0; i<n; i++){
        left.push_back(nums[i]);
        right.push_back(nums[i+n]);
        tot += nums[i] + nums[i+n];
    }
vector<vector<int>>sum1(n+1), sum2(n+1);
    for(int mask =0; mask < (1<<n); mask++){
        int tot1 = 0, tot2 =0, count =0;
        for(int j=0; j<n; j++){
            if(mask & (1<<j)){
                tot1 += left[j];
                tot2 += right[j];
                count += 1;
            }
        }
        sum1[count].push_back(tot1);
        sum2[count].push_back(tot2);
    }
    for(int i=0;i<=n;i++) sort(sum2[i].begin(), sum2[i].end());
    int minDiff = INT_MAX;
    for(int i=0; i<=n; i++) minDiff = min(minDiff, solve(sum1[i], sum2[n-i], tot));
    return minDiff;
}
```

- LC 924: Given network of nodes connected by an adjacency matrix, some nodes are initially infected with malware, which spreads to connected nodes. Find & remove one initially infected node to minimize total no. of infected nodes, & if there are ties, choose the node with the smallest index?

ex: graph = [[1,0,0,0],[0,1,0,0],[0,0,1,1],[0,0,1,1]]
ini = [3,1]
o/p: 3

ex: graph = [[1,1,0],[1,1,0],[0,0,1]]
ini = [0,1]
o/p: 0

A)

```
int findPar(int i, vector<int>& par){
    if(par[i]==i) return i;
    return par[i] = findPar(par[i], par);
}
void merge(int p1, int p2, vector<int>& par, vector<int>& size){
    if(size[p1] > size[p2]){
        par[p2] = p1;
        size[p1] += size[p2];
    } else{
        par[p1] = p2;
        size[p2] += size[p1];
    }
}
int minMalwareSpread(vector<vector<int>>& graph, vector<int>& initial){
    int n = graph.size();
    vector<int> par(n), size(n,1);
    for(int i=0; i<n; i++) par[i] = i;
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            if(graph[i][j]==1){
                int p1 = findPar(i, par), p2 = findPar(j, par);
                if(p1 != p2) merge(p1, p2, par, size);
            }
        }
    }
    vector<int> infected(n);
    for(auto i: initial){
        int p = findPar(i, par);
        infected[p]++;
    }
    int ans = -1, maxSize = -1;
    for(auto i: initial){
        int p = findPar(i, par);
        if(infected[p]==1 and size[p] >= maxSize){
            if(maxSize == size[p]) ans = min(ans, i);
            else ans = i;
            maxSize = size[p];
        }
    }
```

If a graph in a forest has only 1 infected node while others have 2 or more infected nodes, we return that 1 infected node.

```
if(ans == -1){
    int mini = n;
    for(auto i: initial) mini = min(mini, i);
    return mini;
}
return ans;
}
```

- **LC 1755**: Given arr. 'nums' & int. 'goal', find a subsequence whose sum is closest to 'goal'. Return min. possible val. of abs(sum - goal).

ex: nums = [7, -9, 15, -2]
goal = -5
op = 1

ex: nums = [5, -7, 3, 5]
goal = 6
op = 0

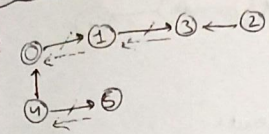
A) (~ meet in the middle)

```
int minAbsDifference(vector<int>& nums, int goal){
    int n = nums.size(), half = nums.size()/2;   // split the array into 2 halves
    vector<int> sum1 = {0}, sum2 = {0};
    for(int i=0; i<half; i++){ // Generate subset sums for the first half
        int sz = sum1.size();
        for(int j=0; j<sz; j++){
            sum1.push_back(sum1[j]+nums[i]);
        }
    }
    for(int i=half; i<n; i++){ // Generate subset sums for the second half
        int sz = sum2.size();
        for(int j=0; j<sz; j++){
            sum2.push_back(sum2[j]+nums[i]);
        }
    }
    sort(sum2.begin(), sum2.end());
    int minDiff = INT_MAX;
    for(int s1 : sum1){
        int target = goal - s1;
        int it = lower_bound(sum2.begin(), sum2.end(), target) - sum2.begin();
        if(it != sum2.size()){
            minDiff = min(minDiff, abs(goal - (s1 + sum2[it])));
        }
        if(it != 0){
            it--;
            minDiff = min(minDiff, abs(goal - (s1 + sum2[it])));
        }
    }
    return minDiff;
}
```

(returns the 1st ele. in sum2 that is not less than the target value.)

(By checking val. at sum2[it-1], we can determine if this ele. gives a smaller abs. diff. compared to the goal than the curr. ele. at sum2[it].)

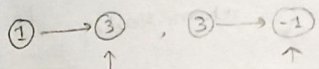
- **LC 1466**: Reorder Routes to Make All paths lead to City Zero

ex: 

ip: n=6, gr = [[0,1],[1,3],[2,3],[4,0],[4,5]]
op: 3
exp: Change the direction of edges shown in dotted such that each node can reach node 0.

A)
```
void dfs(int& cnt, vector<bool>& visited, vector<vector<int>>& gr, int src){
    visited[src] = true;
    for(auto node : gr[src]){
        if(!visited[abs(node)]){
            if(node > 0) cnt++;
            dfs(cnt, visited, gr, abs(node));
        }
    }
}
```

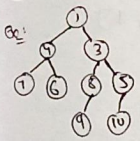
```
int minReorder(int n, vector<vector<int>>& conn){
    int cnt = 0;
    vector<vector<int>> gr(n);
    for(auto node : conn){
        gr[node[0]].push_back(node[1]);
        gr[node[1]].push_back(-node[0]);
    }
    vector<bool> visited(n, false);
    dfs(cnt, visited, gr, 0);
    return cnt;
}
```



(if node > 0, we need to change direction → so cnt++)

(if node < 0 → means we are in correct direction towards 0)

- **LC2471**: Min. no. of operations to sort a binary tree by level (* CYCLE SORTING)

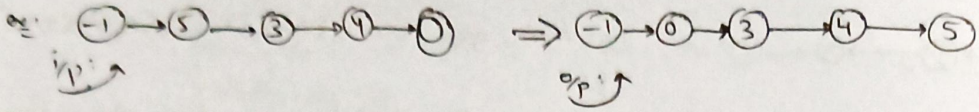
ex: 

op: 3 (swap(4,3), swap(5,7), swap(7,8))

A)
```
int getSwaps(vector<int>& v){
    int n = v.size();
    vector<pair<int,int>> valueIndex(n);
    for(int i=0; i<n; i++){
        valueIndex[i] = {v[i], i};
    }
    sort(valueIndex.begin(), valueIndex.end());
    vector<bool> visited(n, false);
    int swaps = 0;
    for(int i=0; i<n; i++){
        if(visited[i] or valueIndex[i].second == i){
            continue;
        }
        int cycleSize = 0, j = i;
        while(!visited[j]){
            visited[j] = true;
            j = valueIndex[j].second;
            cycleSize++;
        }
        if(cycleSize > 1) swaps += cycleSize - 1;
    }
    return swaps;
}
```

```
int minOps(Node* root){
    // cnt, 'queue'
    // looping/traversing root
    while(q.size() != 0){
        int s = q.size(), i = 0;
        vector<int> v(s);
        while(s--){
            Node* temp = q.front();
            q.pop();
            v[i] = temp->val;
            // pushing child into queue
            i++;
        }
        cnt += getSwaps(v);
    }
    return cnt;
}
```

LC 148 : Sort List




```
Node* sortList(Node* head){
    if(head==NULL or head->next==NULL) return head;
    return merge2LL(head);
}
Node* merge2LL(Node* head){
    Node* midd = findMiddle(head);
    Node* right = midd->next
    midd->next = NULL;
    Node* left = head;
    left= merge2LL(left);
    right = merge2LL(right);
    return merge(left, right);
}
```

(usual merge sort)

(create temp node & now compare 'left' & 'right')

slow - fast pointer (with modification)

```
Node* findMiddle(Node* head){
    Node* slow = head;
    Node* fast = head->next;
    while(fast and fast->next){
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

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

LC 2 : Add Two Numbers

ex: l_1: 9 → 1 → 4 → 9 → 1 → 9 → 9

l_2: 9 → 1 → 9 → 9

o/p: [8, 9, 9, 9, 0, 0, 0, 1]

```
Node* add2Num(Node* l1, Node* l2){
    Node* head =new Node(-1);
    Node* temp = head;
    int carry=0;
    while( l1 and l2){
        int sum = l1->val + l2->val + carry
        int rem = sum%10;
        carry = sum/10;
        Node* t = new Node(rem);
        temp->next = t;
        temp= temp->next;
        l1 = l1->next;
        l2=l2->next;
    }
    while (l1){
        //(similar to above)
    }
    while (l2){
        // (similar to above)
    }
    if (carry){
        Node* t = new Node(carry)
        temp->next=t
        temp=temp->next;
    }
    return head->next;
}
```