RECURSION

Base Probs.:

- i/p - o/p method
  1) Print 1 to n / n to 1
  2) Sort an array/stack
  3) Delete middle ele. in stack
  4) Remove duplicates from a string
  5) Count the no. of occurences
  6) Subset
  7) Permutation → with space (variations)
     → with case change
     → many more ...
  8) Execution in Circle / Josephus Problem
- extended i/p - o/p method
  1) Binary string with no. of 1's greater than no. of 0's
  2) Generate Balanced Parenthesis

Hypothesis - Induction - Base:

- **H:** Assume my func. works correctly for smaller i/p's.
- **I:** Breaking prob. into smaller sub-prob.'s that fits the hypothesis & then combine the results to solve the large problem.
- **B:** Smallest i/p of recursion to avoid TLE

I) i/p - o/p Method

1) **Print 1 to n:** →

```
void print(int n){
    if(n==1){ cout<<n<<" "; return;}
    print(n-1);
    cout<<n<<" ";
}
```

2) Height of a Binary Tree:

Hypo.: height(root) → will give me 'height'

Induc.: height(root->left) → height of left subtree
height(root->right) → " " right "
return 1 + max(lh, rh)

B: root = NULL → return

Algo.:

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

3) Sort an array / stack

We will write for 'stack':

B: if (st.size() == 1) return

→ We will remove top ele. → reduce stack → insert top ele.

Algo:

```
void sort(stack<int> &st){
    if(st.size() == 1) return;
    int temp = st.top();
    st.pop();
    sort(st);
    insert(st, temp);
    return;
}
```

```
void insert(stack<int> &st, int temp){
    if(st.size()==0 or st.top()<=temp){
        st.push(temp);
        return;
    }
    int val = st.top();
    st.pop();
    insert(st, temp);
    st.push(val);
    return;
}
```

for vector → just use vector instead of stack everywhere
(int temp = v[v.size()-1] , v.pop_back(), v.push_back(val))

4) Delete middle element of a stack

B: if(k==1) st.pop(); → means next ele. = middle ele.

Algo:

```
stack<int> midDel(stack<int> s, int size){
    if(s.size()==0) return s;
    solve(s, k);
    return s;
}
```
→ k = size/2 + 1;

```
void solve(stack<int>& s, int k){
    if(k==1){
        s.pop();
        return;
    }
    int temp = s.top();
    s.pop();
    solve(s, k-1);
    s.push(temp);
    return;
}
```

5) Reverse a stack using recursion

Hypo.: insert([1 2 3] [4]) → [1 2 3 4]

smaller i/p → insert([1 2 3] []) → [1 2 3]

Algo:

```
void reverse(stack<int>& st){
    if(st.size()==1) return;
    int ele = st.top();
    st.pop();
    reverse(st);
    insert(st, ele);
}
```

```
void insert(stack<int>& st, int ele){
    if(st.size()==0){ st.push(ele); return;}
    int val = st.top();
    st.pop();
    insert(st, ele);
    st.push(val);
    return;
}
```

6) K^th Symbol in Grammar

Q: Start: 0 → 1st row ; Now if → prev = 0 → then curr = 01
(always) ↳ prev = 1 → " " = 10

2nd row: 01
3rd row: 0110
⋮ ⋮

Given nth row, find Kth symbol. → (1 based indexing)

Hypo: solve(N, K) → gives 0/1
↳ smaller i/p wisely

ex: row (n)
1 - 0
2 - 0 1
3 - 0 1 1 0
4 - 0 1 1 0 | 1 0 0 1

Observations: (N-1) (N) Prev
Same
Complement
Mid → length/2

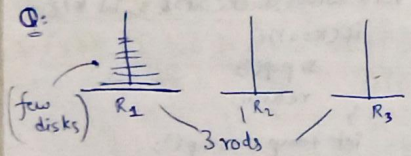
∴ if (K <= mid)
solve(N, K) → solve(N-1, K)
else solve(N, K) → !solve(N-1, K-mid)
↳ (reversing)

∴ Algo:

```
int solve(N, K){
    if(N==1 or K==1) return 0;
    int mid = (pow(2,n-1))/2;
    if(K<=mid) return solve(N-1,K);
    else return !solve(N-1, K-mid);
}
```

7) Tower of Hanoi

Q:



We need to transfer all disks from 1st rod to last rod & the order in last rod must be same as first rod.
{Disks → increasing size → Top 2 bottom}

So → transfer all disks except last one to R_2. $R_1 \to R_3$. Now transfer all disks from $R_2 \to R_3$

∴ Hypo.: solve(n, s→d, h)
small → solve(n, s→h, d);

∴ Algo.:

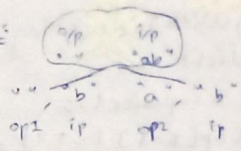
```
long long toh(int n, int from, int to, int aux){
    if(n==1){
        cout<<"Move disk 1 from rod "<<from<<" to rod "<<to<<endl;
        return 1;
    }
    else{
        long long moves = toh(n-1, from, aux, to);
        cout<<"Move disk "<<n<<" from rod "<<from<<" to rod "<<to<<endl;
        moves++;
        moves += toh(n-1, aux, to, from);
        return moves;
    }
}
```

8) Print Subsets / Subsequences (of a string)

Algo.

```
void solve(string ip, string op){
    if(ip.size()==0){
        cout<<op<<endl;
        return;
    }
    string op1 = op, op2 = op;
    op2.push_back(ip[0]);
    ip.erase(ip.begin()+0);
    solve(ip, op1);
    solve(ip, op2);
    return;
}
```

ex:




```
int main(){
    string ip;
    cin>>ip;
    string op="";
    solve(ip, op);
    return 0;
}
```

9) Print Unique Subsets (of a string)

ex: (abc) → ac cax } → In subsequence, order matters, so 'ca' is not a subseq.
(ac ca } → Order does not matter, 'ac', 'ca' are considered as same subset

using a 'set' to print unique;

void so

```
void solve(string ip, string op, set<string>& uniqueSubsets){
    if(ip.size()==0){
        if(uniqueSubsets.find(op)==uniqueSubsets.end()){
            cout<<op<<endl;
            uniqueSubsets.insert(op);
        }
        return;
    }
    //same as above code
}
```

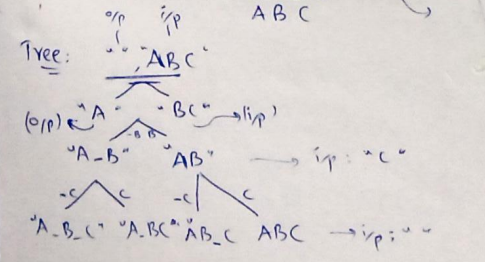
declare in 'int main' (Also same as above)

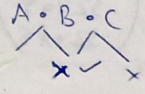
10) Permutation with Spaces

Given → string → generate all strings which have space btw characters

ex: (ABC) → (x)A_B_C(x) ; Note: at start & after end → No space)
A_BC
AB_C
ABC

Tree:





} Choice + decision → recursion

Algo.

```
void stringWithSpace(string ip, string op, vector<string>& ans){
    if(ip.size()==0){
        int z = op.size();
        if(op[z-1] != ' ') ans.push_back(op);
        return;
    }
    string op1 = op, op2 = op;
    op2.push_back(ip[0]);
    op2 += " ";
    op1.push_back(ip[0]);
    ip.erase(ip.begin()+0);
    stringWithSpace(ip, op1, ans);
    stringWithSpace(ip, op2, ans);
    return;
}
```

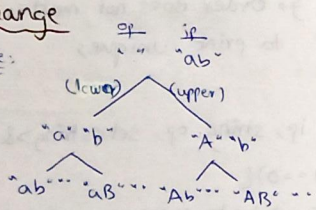
(// Int Main → Same as prev.; just declare 'ans')

11) Permutation With Case Change

ip: ab → op: aB, ab, Ab, AB

(always in lowercase)

tree:



Algo.:

// Changes in above code → op2.push_back(toupper(ip[0]));

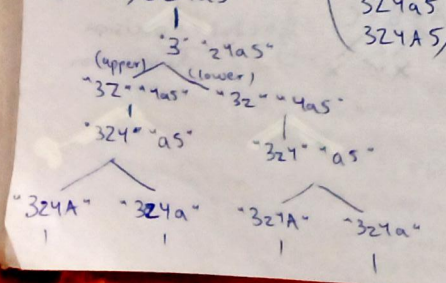
instead of: (op2.push_back(ip[0]); op2+=" ";)

(No 'if' cond. inside "(ip.size()==0)" → so simply push_back in "ans")

12) Letter Case Permutation

ip: 3z4a5 → op: (3z4a5, 3z4A5, 3Z4a5, 3Z4A5) → (prev. ques. + Numbers)

tree:

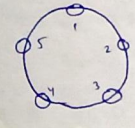


Algo.: (IN INTERVIEW, GIVE PROPER NAME TO FUNC.)

```
void solve(string ip, string op, vector<string>& ans){
    if(ip.size()==0){
        ans.push_back(op);
        return;
    }
    if(isalpha(ip[0])){
        string op1 = op, op2 = op;  op2.push_back(tolower(ch));
        op1.push_back(toupper(ch));  ip.erase(ip.begin()+0);
        solve(ip, op1, ans);  solve(ip, op2, ans);
    }
    else{
        string op1 = op;  op1.push_back(ip[0]);  ip.erase(ip.begin()+0);
        solve(ip, op1, ans);
    } return;
}
```

13) Josephus Problem / Execution in circle

Q: ex: N=5, k=2



There are 'n' people in circle, numbered '1' to 'n'. Starting from '1', you count clockwise & skip 'k-1' people & kill the k^{th} person. Now again counting will begin from person next to 'k' & execution process repeats skipping 'k-1' people. Determine position in initial circle that ensures you are the last person remaining.

We can use 'array' + recursion

ex:

0	1	2	3	4
1	2	3	4	5

If k=2 → you kill 2nd person (index = 1) (erase)

→ so, 'k--' → for '0' based indexing

→ also suppose if k > arr.size() → we use '%' (so that it represents a circle)

Algo.:

```
void solve(vector<int>& person, int k, int index, int& ans){
    if(person.size()==1){
        ans = person[0];
        return;
    }
    index = (index + k) % person.size();
    person.erase(person.begin() + index);
    solve(person, k, index, ans);
}
```

```
int safePos(int n, int k){
    vector<int> person(n);
    for(int i=0; i<n; i++)
        { person[i] = i+1; }
    k = k-1;
    int index = 0, ans = -1;
    solve(person, k, index, ans);
    return ans;
}
```

II) Extended i/p - o/p method

1) Generate all Balanced Parentheses

ex: n=3 → o/p: vector<string> { ((())), (()()), (())(), ()(()), ()()() }

; In normal i/p - o/p → op ip → same datatype

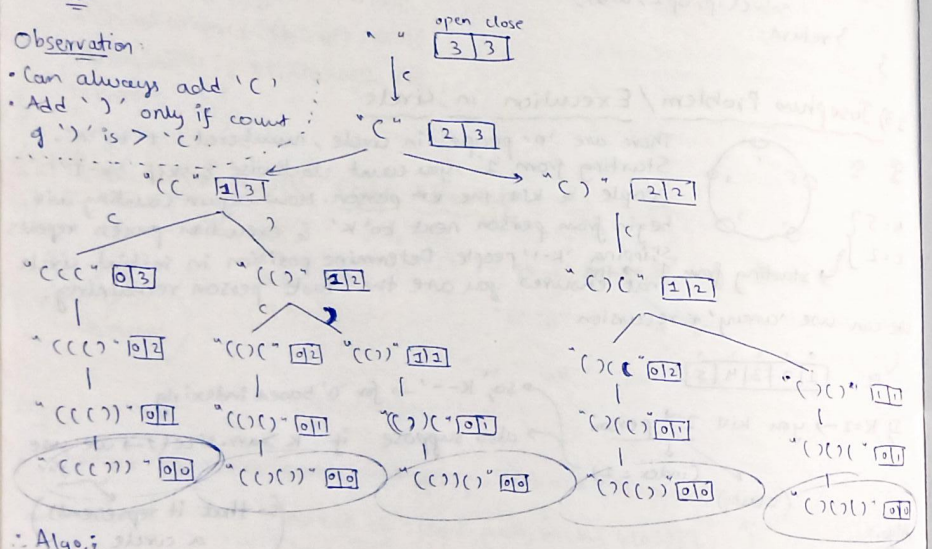
, In extended i/p - o/p → op ip → different datatype
ex: string, int

n=3 → 3 open — need to maintain 2 variables to keep count of brackets
→ 3 close —

)()() → WRONG ✗

∴ Tree:

Observation:
- Can always add '('
- Add ')' only if count of ')' is > '('



∴ Algo.:

```
void solve(int open, int close, vector<string>& ans, string op){
    if(open==0 and close==0){
        ans.push_back(op);
        return;
    }
    if(open!=0){
        string op1=op;      op1.push_back('(');
        solve(open-1, close, ans, op1);
    }
    if(close>open){
        string op2=op; op2.push_back(')');
        solve(open, close-1, ans, op2);
    }
}
```

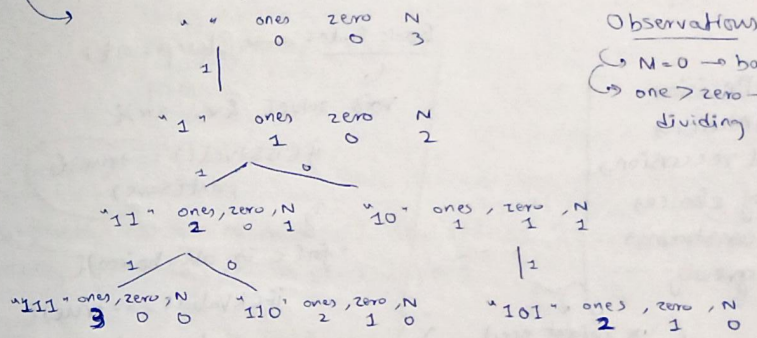
```
//Initializations
open = n → (given)
close = n
temp = ""
```

2) Print N-bit binary no.s having >= 1's than 0's for any prefix

ex: n=3 → o/p: "111" → all it's prefixes: {1, 11, 111} have more 1's than 0's.
(count of '1' can be same as count of '0' also.)
"110" → " " " " {1, 11, 110} " " " " " "
"101" → " " " " {1, 10, 101} " " " " " "

∴ Here: N=3 → one, zeroes: 3 0 / 2 1 / ... / (1 2)
unlike prev. que., here we need to exhaust 'n' directly

Tree:



Observations
- N=0 → base case
- one > zero → then only dividing into 2 sub-tree

∴ Algo.:

```
void solve(int n, string op, vector<int>& ans, int o, int z){
    if(n==0){
        ans.push_back(op);
        return;
    }
    string op1=op;
    op1.push_back('1');
    solve(n-1, op1, ans, o+1, z);
    if(o>z){
        string op2=op;
        op2.push_back('0');
        solve(n-1, op2, ans, o, z+1);
    }
}

vector<string> NBitBinary(int n){
    string op="";
    int o=0, z=0;
    vector<string> ans;
    solve(n, op, ans, 0, z);
    return ans;
}
```

BACKTRACKING

Main Problems of Backtracking:

1) Permutations of a string
2) Largest Number in K^th Swap
3) N digit no. with digits in ↑ order
4) Rat in a Maze
5) Word Break
6) Palindrome Partitioning
7) Combinational Sum
8) Sudoku
9) N Queens

Flow:

1) Choices + Decision
2) All combinations
3) Controlled recursion
4) Number of choices
5) Size of constraints
6) Don't be greedy

(we might need to use 'backtracking')

Ques: 'All Paths' → (if 'all' soln. is asked, then mostly backtracking is used)

(Generally exponential time; so size of constraints will be 'single digit'.
If double digits → might need to use optimized backtracking.)

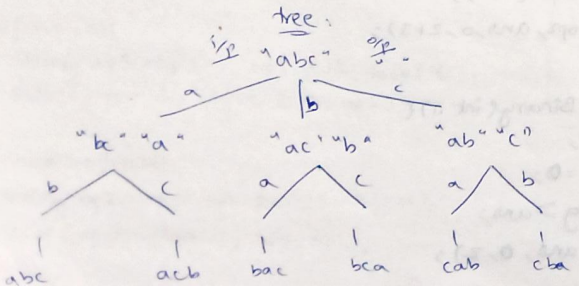
Basic Code: → (Blueprint)

```
Void solve( &v, arr){
    if(isSolved() == true){
        print(save)
    }
    for(c in all choices){
        if(isValid() == true){
            change in v
            func.(v)
            revert changes in v
        }
    }
    return
}
```

I) Permutation of String

Q: Given string 's' - Print all unique permutations.
(might contain duplicates)

Let's see via recursion.



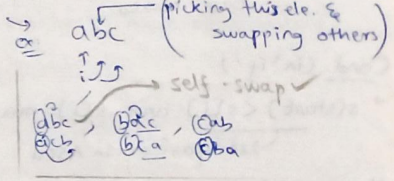
Algo. (recursion):

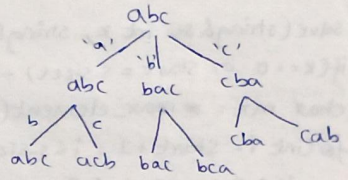
```
void permute (string ip, string op, vector<string>& v){
    if(ip.size()==0){
        v.push_back(op);
        return;
    }
    unordered_set < char > mp;
    for(int i=0; i<ip.size(); i++){
        if(mp.find(ip[i]) == mp.end()){
            mp.insert(ip[i]);
            string newIp = ip.substr(0,i) + ip.substr(i+1);
            string newOp = op + ip[i];
            permute(
        }
    }
    return;
}
```

Now, here instead of making 2 new strings & copy always, we can directly swap & get all permutations

Algo.:

```
void solve(string& s, int start, vector<string>& v){
    if(start == s.size()-1){
        v.push_back(s);
        return;
    }unordered_set<char> mp;
    for(int i=start; i<s.size(); i++){
        if(mp.find(s[i]) == mp.end()){
            mp.insert(s[i]);
            swap(s[start], s[i]);
            solve(s, start+1, v);
            swap(s[start], s[i]);
        }
    }
}
```





T_c: $O(n \cdot n!)$
→ n! permutations & for each permutation, we have 'n' operations.

II) Largest No. in K Swaps

Flow:
1) Prob Stmt. - i/p - o/p
2) Identification - greedy?
3) All choices
4) Backtracking
5) Similarity & dissimilarity to prev. prob.
6) Code
7) T_c

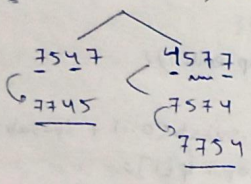
Q: Given 's', do max 'k' swaps to get greatest no.

ex: s = "1234567", k = "4" → "7654321" → done only 3 swaps

If we follow greedy → will not get correct ans. always

(finding largest no. in right & swapping with it)

ex: 4577, k = 2



(4577 → greedy ✗)

If k = 0 → no swaps can be done

ex: 721 → Already greatest no.

i: 0 1 2

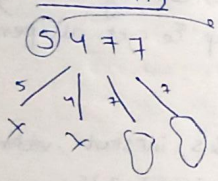
if k = 1 → since no swaps done & 'i' reached to end → so return

base cases

Similarity

In prev. code → ∞ swaps, in this que. → 'k' swaps

Disimilarity



if no. smaller than max. ele. to right, then not going into its recursion

Cond. (in 'if')

s[start] < s[i] and s[i] = max

↳ comparison in ASCII

Algo.:

```
void solve(string&s, int k, string& res, int start){
    if(k==0 or start = s.size()-1) return;  → Base Case
    char mx = *max_element(s.begin()+start+1, s.end());
    for(int i= start+1; i<s.size(); i++){
        if(s[start]< s[i] and s[i]==mx){
            swap(s[start], s[i]);
            if(s.compare(res)>0)  res = s;   (update 'res')
            solve(s,k-1,res,start+1);   (lexographically greater)
            swap(s[start], s[i]);
        }
    }
    solve(s,k,res,start+1);
}
```

→ gets max. 'char'/'ele'

→ $T_c = O(n \times \min(s, res))$

→ if curr. ele. is largest among all, we move to right simply! since no swaps done, so no change in 'k'

```
string findMaxNum(string s, int k){
    string res = str;
    Solve(str, k, res, 0);
    return res;
}
```

Time complexity: Work done = (Work done in 1 node) * (No. of nodes)

$= N^2 * \frac{n(n-1)\dots(n-k+1)}{(n-k)!} \times \frac{(n-k)(n-k-1)\dots 1}{(n-k)! \times 1}$

$= N^2 * \frac{n!}{(n-k)!}$

$\therefore T_c = O\left(n^2 \times \frac{n!}{(n-k)!}\right)$

since we can do atmost 'k' swaps, so node will go only till 'k' level.

III) N digit no.'s with digits in increasing order

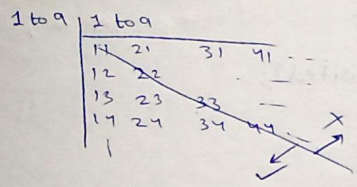
Q: Given 'n', print all 'n' digit no.'s in ↑ order, such that their digits are in strictly ↑ order (from left to right).

ex: n = 2 → op: 12, 13, 14, —, 19, 23, 24, —, 79, 89

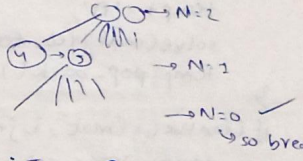
If n = 1 → a single digit is always increasing (as there's no one to compare)

↳ ∴ op: 0, 1, 2, —, 9

for n = 2 (01) ~ 1 → so '0' at start doesn't have any value.



∴ tree:



$\therefore T_c = q^n$

↳ each no. = 'q' choices at max.

Algo.:

```
void solve(int idx, string &str, int n, vector<int>& res){
    if(str.size()==n){
        res.push_back(stoi(str));
        return;
    }
    for(int j=idx+1; j<=9; j++){
        str.push_back(char(j+'0'));
        solve(j, str, n, res);
        str.pop_back();
    }
}
```

```
vector<int> increasingNum(int n){
    string str=""; vector<int>res;
    if(n==1){
        for(int i=0; i<=9; i++)
            res.push_back(i);
        return res;
    }
    else{
        solve(0, str, n, res);
        return res;
    }
}
```

IV) Rat in a Maze

Q rat at '0,0' → reach 'n-1, m-1'. Can move [U] [D] [L] [R] ↑, ↓, ←, →. Cell marked '0' → rat can't go through.
↳ o/p the path: ex: "DDRDRR"

Algo:

```
bool canWeGo(vector<vector<int>>& mat, int i, int j, vector<vector<bool>>& visited){
    if(i<0 or j<0 or i>=mat.size() or j>=mat[0].size()) return false;
    if(mat[i][j]==0 or visited[i][j]) return false;
    return true;
}
void solve(vector<vector<int>>& mat, vector<string>& res, string temp, int i, int j,
           vector<vector<bool>>& visited){
    if(i==mat.size()-1 and j==mat[0].size()-1){
        res.push_back(temp);
        return;
    }
    visited[i][j]=true;
    if(canWeGo(mat,i+1,j,visited)){
        temp+='D';
        solve(mat,res,temp,i+1,j,visited);
        temp.pop_back();
    }
    if(canWeGo(mat,i,j+1,visited)){
        temp+='R'; solve(mat,res,temp,i,j+1,visited); temp.pop_back();
    }
    if(canWeGo(mat,i-1,j,visited)){
        temp+='U'; solve(mat,res,temp,i-1,j,visited); temp.pop_back();
    }
    if(canWeGo(mat,i,j-1,visited)){
        temp+='L'; solve(mat,res,temp,i,j-1,visited); temp.pop_back();
    }
    visited[i][j]=false;
}
vector<string> findPath(vector<vector<int>>& mat){
    vector<string> res;
    if(mat[0][0]==0) return res;
    vector<vector<bool>> visited(mat.size(),vector<bool>(mat[0].size(),false));
    solve(mat,res,"",0,0,visited);
    return res;
}
```

$T_c = O(4^{n^2})$

(Each cell has 4 possible moves)

SLIDING WINDOW ALGORITHM

Identification: subarray + largest size + window size ~ then 'Sliding W/dow'

Main Problems of Sliding Window

- Fixed
  1) Max/Min. subarray of size 'k'
  2) Sum/Avg/(-)ve in every win. size of 'k'
  3) Count occurence of anagram
  4) Max of all subarray of size 'k'
  5) Max of min. for every min. size
- Variable
  a) Largest/Smallest Subarray with Sum 'k'
  b) Largest substring with 'k' distinct characters
  c) Length of largest substring with no repeating characters
  d) Pick toy (similar to 'a')
  e) Min. window substring

1) Fixed

1) Max. Sum Subarray of size 'k'

Q: → return max sum of a subarray of size 'k'.

(i) (j) → window length = (j-i+1), until window len. not reached ↳ keep adding ele.s

Algo:

```
int maxSumSubarray(int k, vector<int>& arr, int n){
    int ans=0, i=0, j=0, sum=0;
    while(j<n){
        if((j-i+1)<k){
            sum+=arr[j];
            j++;
        }
        else if((j-i+1)==k){
            sum+=arr[j];
            ans=max(ans,sum);
            i++;
            j++;
            sum-=arr[i-1];
            if(j==N) break;
        }
    }
    return ans;
}
```

2) 1st -ve no. in every window of size 'k'

↳ Using a queue to maintain the -ve no.s coming & going.

Algo.:

```
vector<int> firstNegInt(vector<int>&a, int n, int k){
    vector<int> negative;
    queue<int> q;
    int i=0, j=0;
    while(j<n){
        if((j-i+1)<k){
            if(a[j]<0) q.push(a[j]);
            j++;
        }
        else if((j-i+1)==k){
            if(a[i]<0) q.push(a[j]);
            if(!q.empty()) negative.push_back(q.front());
            else negative.push_back(0);
            if(!q.empty() and a[i]==q.front()) q.pop();
            i++; j++;
        }
    }
    return negative;
}
```

3) Count Occurences of Anagrams

Q: Given word 'pat' & text 'txt'. return count of occurences of anagrams of word in text.

eg: forxxorfxdofr = txt } → 3 → exp: (for, orf, ofr)
pat = for

∴ Basically we need freq of chars
We compare window freq. & 'pat' freq & return accordingly

Algo.:

```
int search(string pat, string txt){
    map<char,int> mp1, mp2;
    for(char ch: pat) mp1[ch]++;
    int i=0, j=0, k=pat.size(), ans=0;
    while(j<txt.size()){
        if((j-i+1)<k){
            mp2[txt[j]]++;
            j++;
        }
        else if((j-i+1)==k){
            mp2[txt[j]]++;
            bool match=true;
            for(auto entry: mp1){
                if(mp2[entry.first]!=entry.second){
                    match=false;
                    break;
                }
            }
            if(match) ans++;
            mp2[txt[i]]--;
            i++;
            j++;
        }
    }
    return ans;
}
```

(Add curr. char. to mp2)

4) Maximum of all subarrays of size 'k'

eg: 1 3 -1 -3 5 3 6 7 → 3 3 5 5 6 7 → Need to return this vector

(max. ele) → So we can add this in our priority queue → (max heap)

(because smaller values won't be our ans., so we use prio. queue)

check → if 'top' ele. not in window → so pop

Algo.:

```
vector<int> slidingMax(vector<int>&a, int k){
    vector<int> ans;
    priority_queue<pair<int,int>> pq;
    if(k>a.size()){
        ans.push_back(*max_element(a.begin(),a.end()));
        return ans;
    }
    int i=0, j=0;
    while(j<a.size()){
        if((j-i+1)<k){
            pq.push({a[j], j});
            j++;
        }
        else if((j-i+1)==k){
            pq.push({a[j], j});
            while(pq.top().second<i) pq.pop();
            ans.push_back(pq.top().first);
            i++; j++;
        }
    }
    return ans;
}
```

II) Variable

General format of fixed size & variable size; Identification

Fixed Size

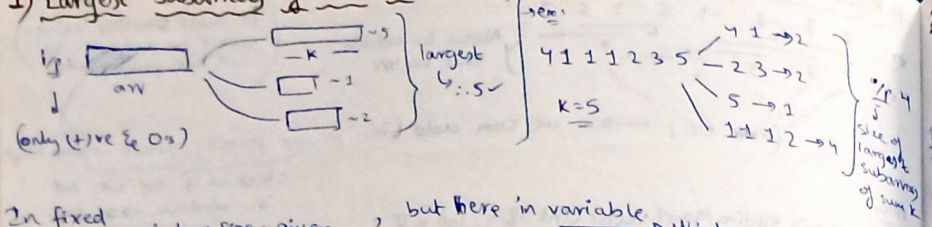
```
while(j<size){
    // calculations → [repeated code which were written in 'if' & 'else if']
    if(win.size<k)
        j++;
    else if(win.size==k){
        ans ← calc.
        calc. → remove i
        (win. size maintain & slide)
    }
}
```

Variable Size

```
while(j<size){
    // calculations
    if(cond.<k)
        j++
    else if(cond.==k){
        ans ← calculations
        j++;
    }
    else if(cond.>k){
        while(cond.>k){
            // remove calc. for i
            i++
        }
        j++
    }
}
return ans;
```

→ given array/string
→ need subarray (or) substring
→ k → sum (or) cond. (given to find win. size)

1) Largest Subarray of Sum K



In fixed → window size = given, Sum = maximise ; but here in variable → Window size = maximise, sum = given

(*This algo. doesn't work for (-)ve no.'s)

∴ Algo:

```
sum = 0
while (j < size()){
    sum = sum + arr[j]; → //calculation
    if (sum < k) j++;
    else if (sum == k){
        if((j-i+1)>mx) mx = j-i+1;   (OR)  mx = max(mx, j-i+1);
        j++;
    }
    else if (sum > k) {
        while (sum > k){
            sum = sum - arr[i];
            i++;
        }
        j++;   ← // Add → if(sum == k)  mx = max(j-i+1, mx);
    }
}
return mx;
```

2) Longest Substring With k Unique Characters

ex: aabacbebebe , k=3 ⇝ o/p: 7 (cbebebe)

↳ We can use a map to store unique char. → mp.size() < k → add; = k → find mx' length; > k → do i++, until it's not <= to 'k' again

Algo:

```
int longestKSubstr(string s, int k){
    map<char,int> mp;
    int i=0, j=0, mx=-1;
    while (j < s.size()){
        mp[s[j]]++;
        if(mp.size() < k) j++;
        else if(mp.size() == k){
            mx = max(mx, j-i+1);
            j++;
        }
        else if(mp.size() > k){
            while(mp.size() > k){
                mp[s[i]]--;
                if(mp[s[i]] == 0) mp.erase(s[i]);
                i++;
            }
            if(mp.size() == k) mx = max(mx, j-i+1);
            j++;
        }
    }
    return mx;
}
```

3) Longest Substring Without Repeating Characters

ex: pwwkew → o/p: 3 → {kew}

↳ (No repeating characters) == (All unique characters) ↳ (similar to prev. ques.)

Identification → we have 'string' → we need substring length → condn. ⇒ for window (all unique chars.) → This q. satisfies, so we can apply Sliding Window ✓

Prev. → mp.size() == k, in this que., mp.size() == ? (Think...)

Ans: mp.size() == j-i+1 } → since we need all unique characters, so atmost unique chars. length == sliding window length.

↳ //Rest all same as prev. code

Algo:

```
int lengthOfLongSubstr(string s){
    int i=0, j=0, mx=0;   unordered_map<char,int> mp;
    while(j < s.size()){
        mp[s[j]]++;
        if(mp.size() == (j-i+1)){
            mx = max(j-i+1);
            j++;
        }
        else if(mp.size() < (j-i+1)){
            while(mp.size() < (j-i+1)){
                mp[s[i]]--;
                if(mp[s[i]] == 0) mp.erase(s[i]);
                i++;
            }
            if(mp.size() == (j-i+1)) mx = max(mx, j-i+1);
            j++;
        }
    }
    return mx;
};
```

Note:
mp.size() > (j-i+1) → not possible because atmax, map will have size = window length, it can't go more than that → if it goes, then nice → as we need all unique characters

4) Minimum Window Substring

Q: Given 2 strings 's' & 't' of len. 'm' & 'n'. Return min. window substr. of 's' such that every char. in 't' (including duplicates) is included in the window. If no such substr, return "".

Ip: s = "ADOBECODEBANC", t = "ABC" → o/p: BANC

↓

Means, I need 1A, 1B, 1C] in 's' (presence is imp., not order) ↳ need to find smallest substr which has all characters.

ex: S = TOTMTAPTAT
t = TTA

↳ in map: | T 2 | A 1 |

, count = t.size()
↳ (will tell, no. of chars. we are left to match)
(count = 0 → update min. win. len.)

count = 3 2 1 0

	0	1	2	3	4	5	6	7	8	9
	T	O	T	M	T	A	P	T	A	T
i	↑									
j	↑	→	→	→	→	→	→	→	→	→

(index) (0) (2) (5)

↳ arr[j] → present in map
↓
(decrease count, decrease that char. freq.)

At 4th index, count of T = 0, but we will still ↓ it
↳ ∴ mp['T'] = -1

T	2 → 1 → 0
A	1

map

↓
not present → simply, move on.

↳ reason:

			T	T	A

not needed → as we need T = 2 only

(so '-1' denotes we got an extra 'T')

(prev. potential answers)

Algo.:

```
string minWindow(string s, string t){
    if(t.size() > s.size()) return "";
    unordered_map<char,int> mp;
    int i=0, j=0, start=(-1), len = INT_MAX;
    for(char ch: t) mp[ch]++;
    int count = t.size();
    while(j < s.size()){
        if(mp.find(s[j]) != mp.end()){ → ele. exists in map
            if(mp[s[j]] > 0) count--;
            mp[s[j]]--; → decrease count of curr. char.
        }
        while(count == 0){
            if(len > (j-i+1)){ // updating window size
                len = j-i+1;
                start = i;
            }
            if(mp.find(s[i]) != mp.end()){
                mp[s[i]]++;
                if(mp[s[i]] > 0) count++;
            }
            i++;
        }
        j++;
    }
    if(start != -1) return s.substr(start, len);
    return "";
}
```

DP

VI) DP on LIS

1) Length of LIS: ex: nums = [10,9,2,5,3,7,101,18] → op: 4 ([2,3,7,101])

nums = [7,7,7,7,7] → op: 1

Soln. 1: $T_c = O(n^2)$

```
int lengthOfLIS(vector<int>& nums){
    vector<int> dp(2505, -1);
    dp[0] = 1;
    int ans = INT_MIN;
    for(int i=0; i<nums.size(); i++){
        for(int j=0; j<=i-1; j++){
            if(nums[j] < nums[i]){
                dp[i] = max(dp[i], 1+dp[j]);
            }
        }
        if(dp[i] == -1) dp[i] = 1;
        ans = max(ans, dp[i]);
    }
    return ans;
}
```

[10, 9, 2, 5, 3, 7, 101, 18]
dp: 1 1 1 2 2 3 (i)

we compare from start & check if any ele. < 3

(if yes) → max(length of LIS at that index + 1 , curr. length of LIS at curr. index)

(i) → length of longest ↑ subsequence ending at 'i'.

Soln. 2: $T_c = O(n \log n)$

```
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();
}
```

ex: 2, 5, 3, 7, 11, 9, 10, 13, 6, 8, 9, 10, 12

'v': 2 ~~5~~ ~~7~~ ~~11~~ ~~10~~ ~~13~~ 12
3 6 ~~9~~ 9 10
8

↳ LIS array is either same (or) increasing only

2) Printing LIS:

```
vector<int> printLIS(vector<int>& nums, int n){
    vector<int> dp(n,1);
    vector<int> hash(n,1);
    for(int i=0; i<n; i++){
        hash[i] = i;
        for(int j=0 ; j<i ; j++){
            if(nums[j] < nums[i]){
                if(1+dp[j] > dp[i]){
                    dp[i] = 1+dp[j];
                    hash[i] = j;
                }
            }
        }
    }
    int ans = -1, lastIndex = -1;
    for(int i=0; i<n; i++){
        if(dp[i] > ans){
            ans = dp[i];
            lastIndex = i;
        }
    }
    vector<int> temp;
    temp.push_back(nums[lastIndex]);
    while(hash[lastIndex] != lastIndex){
        lastIndex = hash[lastIndex];
        temp.push_back(nums[lastIndex]);
    }
    reverse(temp.begin(), temp.end());
    return temp;
}
```

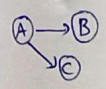
GRAPH

Main / Parent Problems of Graph:

1) Types of graphs
2) How to implement?
3) Traversing a graph → BFS, DFS
4) Cycle detection
5) Topological Sort → Kahn's Algo., DFS Based
6) Flood Fill
7) Connected components
8) Shortest Path Algo. → Single source (3 algo.'s), multi source
9) DSU
10) MST → Prim's, Kruskal's
11) Hamiltonian Path
12) Graph Colouring
13) SCC → Kosaraju's Algo.
14) Network Flow → Ford Fulkerson, Edmonds Karp

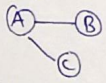
I) Types of Graphs

* Directed ⇓



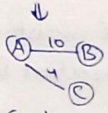
+ (unweighted)

* Undirected ⇓



{bidirectional} + (unweighted)

* Weighted ⇓

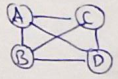


(undirected + weighted)

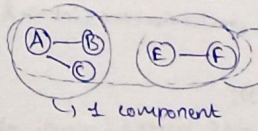
* Unweighted

* Complete Graph
* Disconnected graph (forest type)
* Euler graph
* Hamiltonian graph
* Pseudo graph
* Null graph
* Cyclic graph
* Acyclic graph

⇒ Complete Graph: every node is connected to every other node of the graph ~ ex:

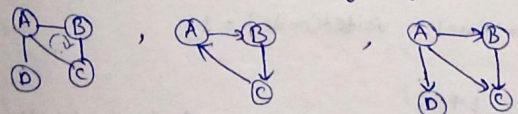


⇒ Disconnected Graph: ex:

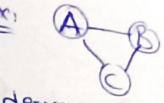


⇒ Cyclic Graph: graph containing cycle

ex:

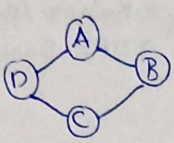


Acyclic graph

⇒ Euler Graph : degree of each node is even. ex: 

degree: A=2, B=2, C=2

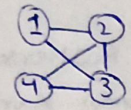
⇒ Hamiltonian Graph: every node being covered only once except the start node.

↳ ex: 

visit → A=2
B=1
C=1
D=1

II) Graph Implementation

1) Adjacency Matrix

↳ ex: 

≡

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

2) Adjacency List ↳ ex:

≡
1 → 2,3
2 → 1,3,4
3 → 1,4,2
4 → 2,3

• Traversal → Equidistant = BFS
↳ Direction Based = DFS

BFS: Algo.:
↓
$T_C = O(V+E)$
$S_C = O(V)$

queue → source node
while queue is not empty:
f → front node.
traverse neighbours of 'f':
↳ unvisited
↳ 1) push it to queue
2) Mark visited

For DFS:
→ $T_C = O(V+E)$
$S_C = O(V+E)$

(*Refer Main Notes for above)

III) Find if Path exists

```
void dfs(vector<int>& visited, vector<vector<int>>& graph, int node){
    visited[node] = 1;
    for(int nbr : graph[node]){
        if(!visited[nbr]) dfs(visited, graph, nbr);
    }
}
```
} DFS

```
void bfs(..., ..., ...){
    queue<int> q; q.push(node); visited[node] = 1;
    while(!q.empty()){
        int f = q.front(); q.pop();
        for(int nbr : graph[f]){ if(!visited[nbr]) {q.push(nbr); visited[nbr]=1;} }
    }
}
```
} BFS