N-Queen

```
bool isQueenSafe(vector<vector<int>> chess, int row, int col){
    for(int i=row-1; i>=0; i--){                                    //Colums check
        if(chess[i][col]==1)
            return false;
    }
    for(int i=row-1, j=col-1; i>=0 && j>=0; i--,j--){               //Principle diagonal check
        if(chess[i][j]==1)
            return false;
    }
    for(int i=row-1, j=col+1; i>=0 && j<chess.size(); i--,j++){     //Secondary diagonal check
        if(chess[i][j]==1)
            return false;
    }
    return true;
}

void printNqueens(vector <vector<int>> chess, string qsf,int row){
    if(row==chess.size()){
        cout<<qsf+"."<<endl;
        return;
    }
    for(int col=0; col<chess[0].size(); col++){
        if(isQueenSafe(chess,row,col)){
            chess[row][col]=1;
            printNqueens(chess, qsf+to_string(row)+"-"+to_string(col)+ ", ", row+1);
            chess[row][col]=0;
        }
    }
}
```

```
void nQueen(int n, int row,vector<bool> &col, vector<bool> &pd, vector<bool> &sd, string asf){
    if(row == n){
        cout << asf + "." << endl;
        return;
    }
    for(int i = 0; i < n; i++){
        if(col[i] != true && pd[row + i] != true && sd[row-i+n-1] != true){
            col[i] = true;
            pd[row + i] = true; //row+col = principle diag
            sd[row - i + n - 1] = true;

            nQueen(n, row + 1, col, pd, sd, asf + to_string(row) + "-" + to_string(i) + ", ");

            col[i] = false;
            pd[row + i] = false;
            sd[row - i + n - 1] = false; //row-col+(boardSize-1)
        }
    }
}
```

Sudoko Solver

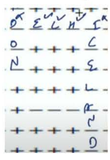
```
bool isValid(vector<vector<int>> sudoko, int r, int c, int data){
    //Row check
    for(int i = 0; i < 9; i++){
        if(sudoko[i][c] == data)
            return false;
    }
    //Col check
    for(int i = 0; i < 9; i++){
        if(sudoko[r][i] == data)
            return false;
    }
    //Box check
    int sr = r - r%3;
    int sc = c - c%3;
    for(int i = sr; i < sr+3; i++){
        for(int j = sc; j < sc+3; j++){
            if(sudoko[i][j] == data)
                return false;
        }
    }
    return true;
}
```

```
void sudokoSolver(vector<vector<int>> sudoko, int i, int j){
    if(i == 9){
        display(sudoko);
        return;
    }
    int ni{}, nj{};
    if(j == 8){
        ni = i+1;
        nj = 0;
    }
    else{
        ni = i;
        nj = j+1;
    }
    if(sudoko[i][j] != 0)
            sudokoSolver(sudoko, ni, nj);
    else{
        for(int po = 1; po <= 9; po++){ //Possible options
            if(isValid(sudoko, i, j, po)){
                sudoko[i][j] = po;
                sudokoSolver(sudoko, ni, nj);
                sudoko[i][j] = 0;
            }
        }
    }
}
```

Crossword

```
bool canPlacewordHorizontally(vector<vector<char>> board, string word, int sr, int sc){
    /*
    1. No of empty slots = No of letters in word
    2. left and right of word must be +
    3. Word must correctly fit into the empty slot
    */
    if(sc-1 >= 0 && board[sr][sc-1] != '+') //Left side hain and vaha par + nahi pada
        return false;
    if(sc + word.size() < board[0].size() && board[sr][sc+word.size()] != '+') //Right side hain and vaha par + nahi pada
        return false;
    if(sc + word.size() > board[0].size()) //If board size exceeds the no of - spots available
        return false;
    for(int x = 0; x < word.size(); x++){
        if(board[sr][sc+x] == '-' || board[sr][sc+x] == word[x])
            continue;
        else
            return false;
    }
    return true;
}

bool canPlacewordVertically(vector<vector<char>> board, string word, int sr, int sc){
    /*
    1. No of empty slots = No of letters in word
    2. top  and bottom of word must be +
    3. Word must correctly fit into the empty slot
    */
    if(sr-1 >= 0 && board[sr-1][sc] == '-')
        return false;
    if(sr + word.size() < board.size() && board[sr+word.size()][sc] != '+')
        return false;
    if(sr + word.size() > board.size())
        return false;
    for(int x = 0; x < word.size(); x++){
        if(board[sr+x][sc] == '-' || board[sr+x][sc] == word[x])
            continue;
        else
            return false;
    }
    return true;
}

vector<bool> placewordHorizontally(vector<vector<char>> &board, string word, int sr, int sc){
    vector<bool> placed(word.size(), false);
    for(int i = 0; i < word.size(); i++){
        if(board[sr][sc+i] == '-'){
            board[sr][sc+i] = word[i];
            placed[i] = true;
        }
    }
    return placed;
}

void unplacewordHorizontally(vector<vector<char>> &board, vector<bool> placed, int sr, int sc){
    for(int i = 0; i <placed.size(); i++){
        if(placed[i])
            board[sr][sc+i] = '-';
    }
}

vector<bool> placewordVertically(vector<vector<char>> &board, string word, int sr, int sc){
    vector<bool> placed(word.size(), false);
    for(int i = 0; i < word.size(); i++){
        if(board[sr+i][sc] == '-'){
            board[sr+i][sc] = word[i];
            placed[i] = true;
        }
    }
    return placed;
}

void unplacewordVertically(vector<vector<char>> &board, vector<bool> placed, int sr, int sc){
    for(int i = 0; i <placed.size(); i++){
        if(placed[i])
            board[sr+i][sc] = '-';
    }
}
//* Level - Words
//* Option- Where to place
void crossWord(vector<vector<char>> board, vector<string> words, int vidx){
    if(vidx == words.size()){
        display(board);
        return;
    }
    string word =  words [vidx];
    for (int i = 0; i < board.size(); i++){
        for (int j = 0; j < board.size(); j++){
            if(board[i][j] == '-' || board[i][j] == word[0]){
                if (canPlacewordHorizontally(board, word, i, j)){
                    vector<bool> placed = placewordHorizontally(board, word, i, j);
                    crossWord (board, words, vidx + 1);
                    unplacewordHorizontally(board, placed, i, j);
                }
            }
            if(canPlacewordVertically(board,word, i, j)){
                vector<bool> placed = placewordVertically(board, word, i, j);
                crossWord (board, words, vidx + 1);
                unplacewordVertically(board,placed, i, j);
            }
        }
    }
}
```

vector<bool> placed is used in placeHorizontally, placeVertically because it while unplacing the placed characters whilc backtracking, we want to unplace onlyt those charactes placed in current iteration ythat we mean that



Here D of DON and I of IRELAND are already placed, so I have placed only ELH of DELHI currently, so while unplacing the characters, I cannot unplace D and i

Partition in K Subsets

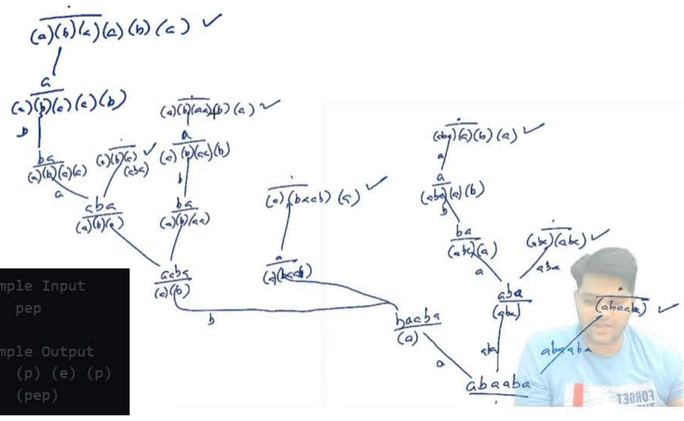
```
Level -> Number btw 1 and n
Option -> 1. To start new subset (increases no of non empty sets)
          2. To be a part of an already existing non-empty set(no increase in no of non empty sets)
Storage -> 2d Vector of k rows is used to store a subset on each row
void Kpartition(int i, int n, int k, int NonEmpSet, vector<vector<int>> vec){
    if(i > n){
        static int count = 1;
        if(NonEmpSet == k){
            cout << count << ". ";
            count++;
            for(int x = 0; x < vec.size(); x++){
                cout << "[";
                for(int y = 0; y < vec[x].size(); y++){
                    if(y == vec[x].size()-1)
                        cout << vec[x][y];
                    else
                        cout << vec[x][y] << ", ";
                }
                cout <<  "] ";
            }
            cout << endl;
        }
        return;
    }
    for(int j = 0; j < vec.size(); j++){
        if(vec[j].size() == 0){ //Start a new subset
            vec[j].push_back(i);
            Kpartition(i+1, n, k, NonEmpSet+1, vec);
            vec[j].pop_back();
            break;
        }
        else{
            vec[j].push_back(i); //Append to an already existing subset
            Kpartition(i+1, n, k, NonEmpSet, vec);
            vec[j].pop_back();
        }
    }
}
```

```
Sample Input
    3
    2

Sample Output
    1. [1, 2] [3]
    2. [1, 3] [2]
    3. [1] [2, 3]
*/
```

Palindromic Partition

```
bool isPalindrome(string str){
    string temp = str;
    reverse(str.begin(), str.end());
    return str == temp;
}

void solution(string str, string asf){
    if(str.size() == 0){
        cout << asf << endl;
        return;
    }
    for(int i = 0; i < str.size(); i++){
        string temp = str.substr(0, i+1);
        if(isPalindrome(temp)){
            string ros = str.substr(i+1);
            solution(ros, asf + '(' + temp + ") ");
        }
    }
}
```

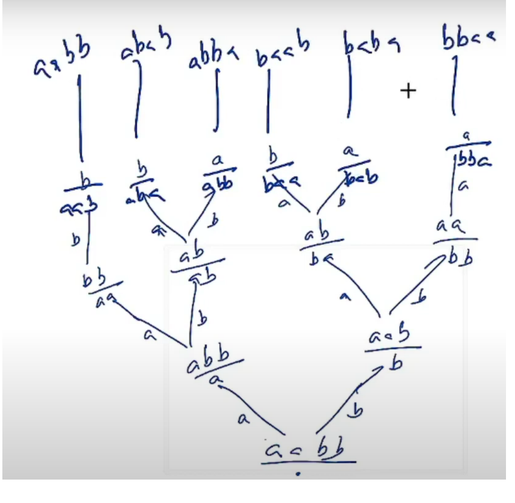
```
Sample Input
    pep

Sample Output
    (p) (e) (p)
    (pep)
*/
```



Palindromic Permutation

Print all permutations of a string which are palindromic

```
 Approach
 1. Find the frequency of each alphabet in given string
 2. Divide the frequency of each by 2 and find the aplhabets with odd frequench
 3. Now generate all substings of string(with halp the frequency of alphabets)
 5. If no of odd freq character is >1, then palincromic permutations cannot be printed
 6. print the string + oddChar + reverse(string) -> palindromic permutation

void palindromicPermutation(int i, int len, map<char, int> mp, char odd, string asf){
    if(i > len){
        string temp = asf;
        reverse(asf.begin(), asf.end());
        cout << temp + odd + asf << endl;
        return;
    }
    for(auto &p: mp){
        if(p.second > 0){
            p.second --;
            palindromicPermutation(i+1, len, mp, odd, asf + p.first);
            p.second++;
        }
    }
}
int main(){
    string str;
    cin >> str;
    map<char, int> mp;
    for(char &ch: str){
        if(mp.find(ch) == mp.end())
            mp[ch] = 1;
        else
            mp[ch]++;
    }
    int odds{}, length{};
    char odd;
    for(auto &p: mp){
        if(p.second % 2 == 1){
            odds++;
            odd = p.first;
        }
        p.second /= 2;
        length += p.second;
    }
    palindromicPermutation(1, length, mp, odd, "");
}
```



Partition into two subsets with minimum sum difference

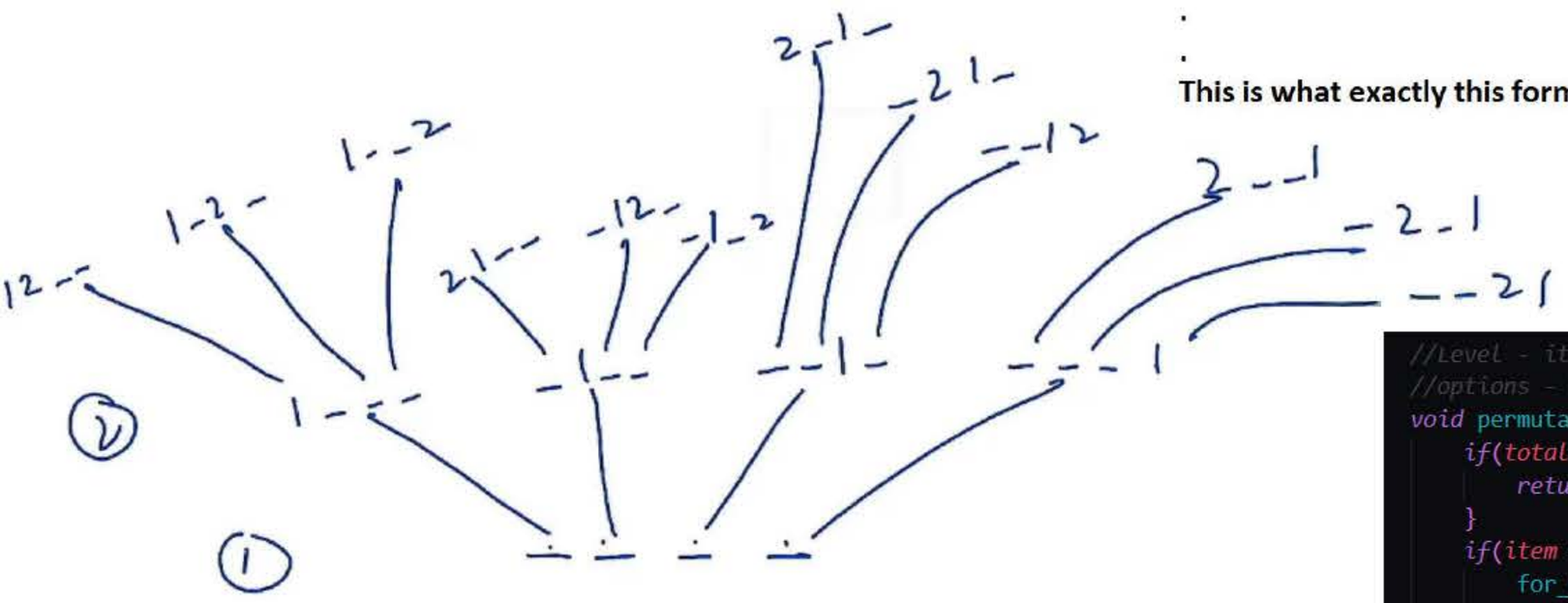
```
Level - element ie vec[i]
Option - Can go to left or right subset if size permits
string ans;
void tugOfWar(vector<int> vec, int i, vector<int> left, vector<int> right, int sumOfLeft, int sumOfRight){
    if(i == vec.size()){
        if(left.size() == right.size() || abs(int(left.size()-right.size())) == 1){
            static int mn = INT_MAX;
            int diff = abs(sumOfLeft - sumOfRight);
            if(diff < mn){
                mn = diff;
                ans = answer(left, right);
            }
        }
        return;
    }
    if(left.size() < (vec.size()+1) / 2){
        left.push_back(vec[i]);
        tugOfWar(vec, i+1, left, right, sumOfLeft + vec[i], sumOfRight);
        left.pop_back();
    }

    if(right.size() < (vec.size()+1) / 2){
        right.push_back(vec[i]);
        tugOfWar(vec, i+1, left, right, sumOfLeft, sumOfRight + vec[i]);
        right.pop_back();
    }
}
```

You have to divide these n integers into 2 subsets such that the difference of sum of two subsets is as minimum as possible.
If n is even, both set will contain exactly n/2 elements. If n is odd, one set will contain (n-1)/2 and other set will contain (n+1)/2 elements.

PERMUTATION

$$ {}^{n}P_{r} = \frac{n!}{(n-r)!} = \frac{n \times n-1 \times n-2 \cdots (n-(r-1))\,(n-r)!}{(n-r)!} \rightarrow \frac{4!}{2!} = \frac{4 \cdot 3 \cdot 2!}{2!} $$

at each level, options for item keeps reducing
first item - has n options
second item - has n-1 options
.
.
This is what exactly this formula says



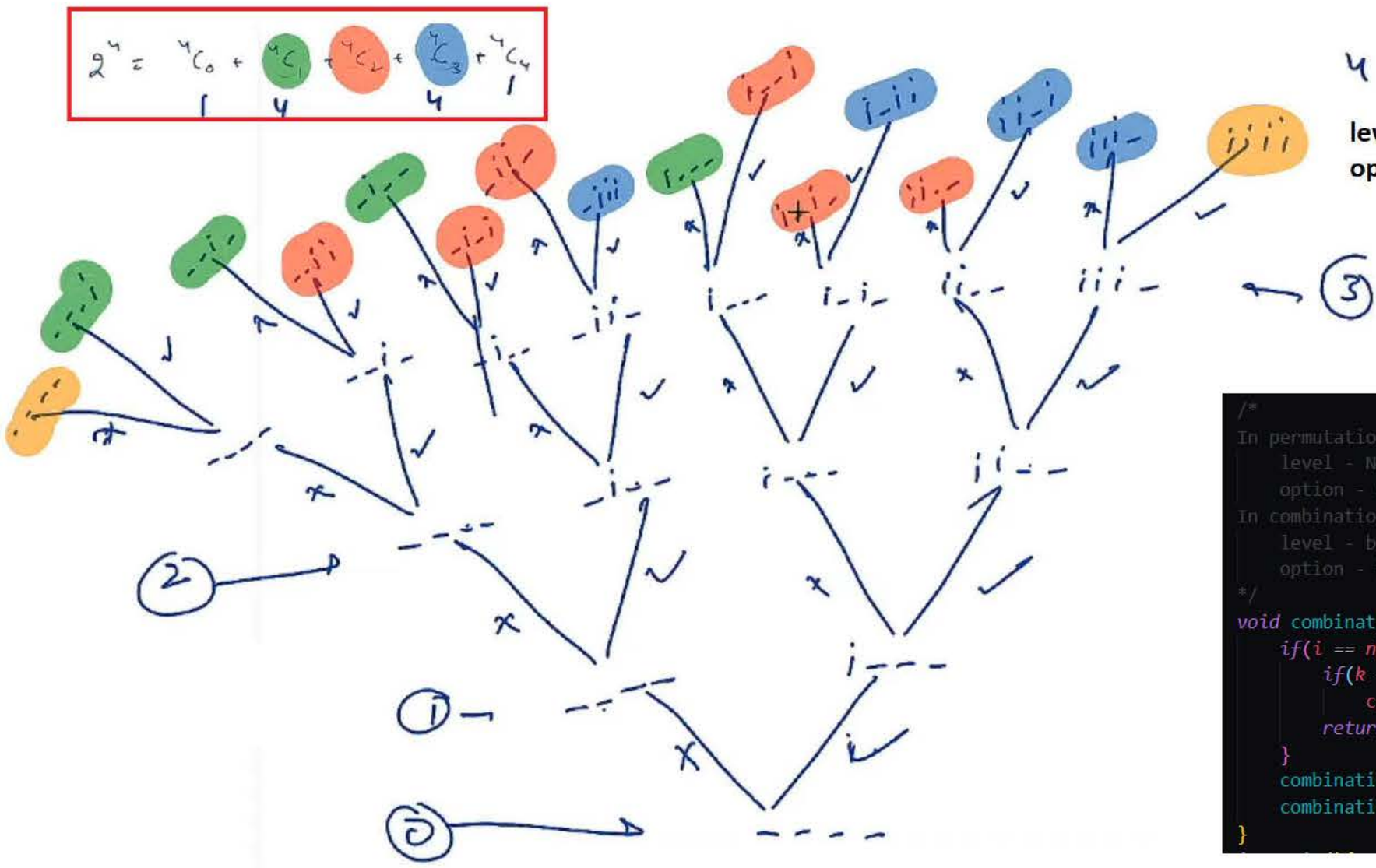
level - item
option - box

```
//Level - item
//options - box
void permutations(vector<int> boxes, int item, int totalItem){
    if(totalItem > boxes.size()){
        return;
    }
    if(item == totalItem+1){
        for_each(boxes.begin(), boxes.end(), [](int x){cout << x;});
        cout << endl;
        return;
    }
    for(int box = 0; box < boxes.size(); box++){
        if(boxes[box] == 0){
            boxes[box] = item;
            permutations(boxes, item+1, totalItem);
            boxes[box] = 0;
        }
    }
}
```

COMBINATION

1. You are give a number of boxes (nboxes) and number of identical items (ritems).
2. You are required to place the items in those boxes and print all such configurations possible.

This is equicvalent to selecting
2 boxes out of 4

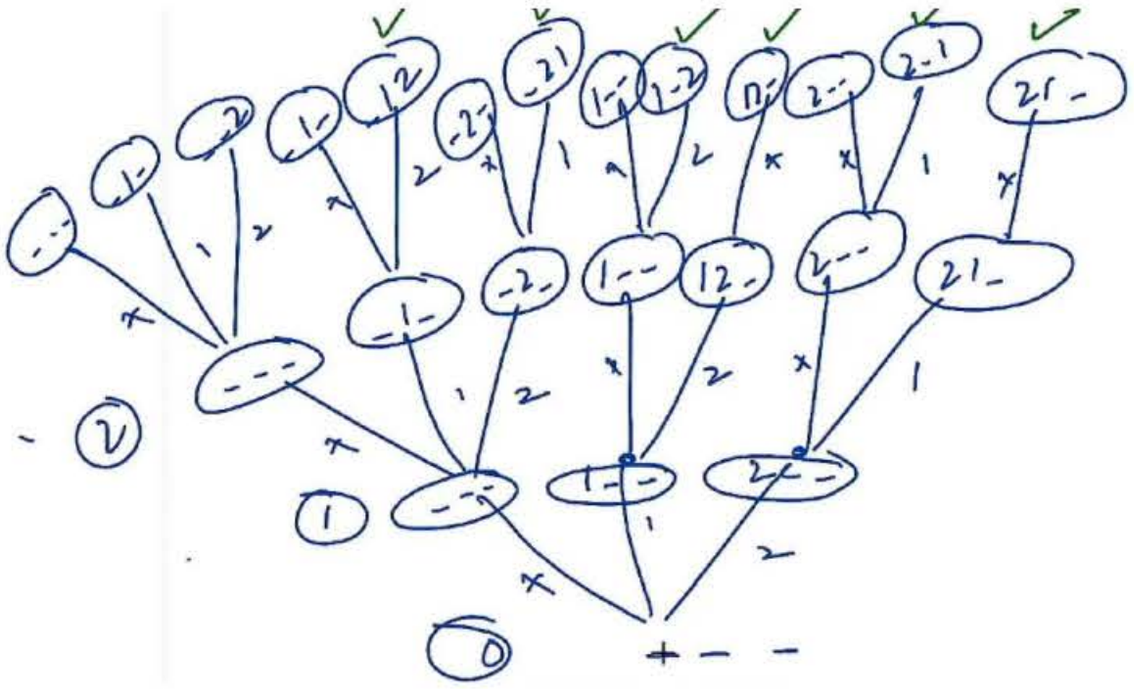


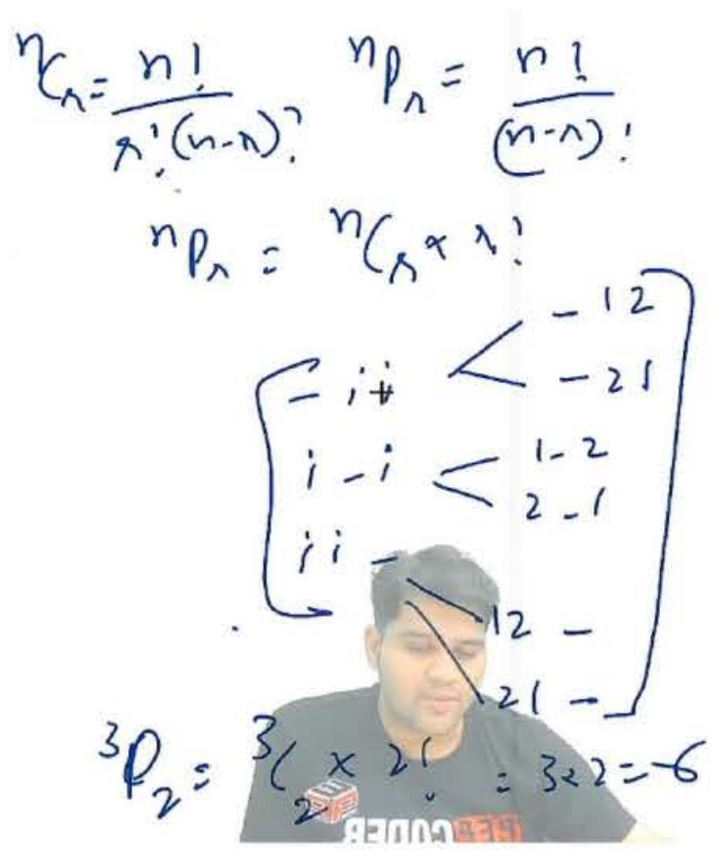
level - box no
option - item ayega/nahi ayega

```
/*
In permutation
    level - Number(item)
    option - which box to place
In combination
    level - box
    option - to place item or not
*/
void combinations(string str, int i, int n, int k){
    if(i == n){
        if(k == 0)
            cout << str << endl;
        return;
    }
    combinations(str + "i", i+1, n, k-1);
    combinations(str + "-", i+1, n, k);
}
```

PERMUTATION 2

In permutation-1 we used pure permutation
level-item
Option - in which box to place
In this problem, we are using combination and then arranging them to get permutation
ie nPr = nCr x r!
level-box
option - Item ayega/nahi-ayega (AS IN COMBINATION)
Agar ayega to konsa item ayega(GENERATING PERMUTATION FROM COMBINATION)

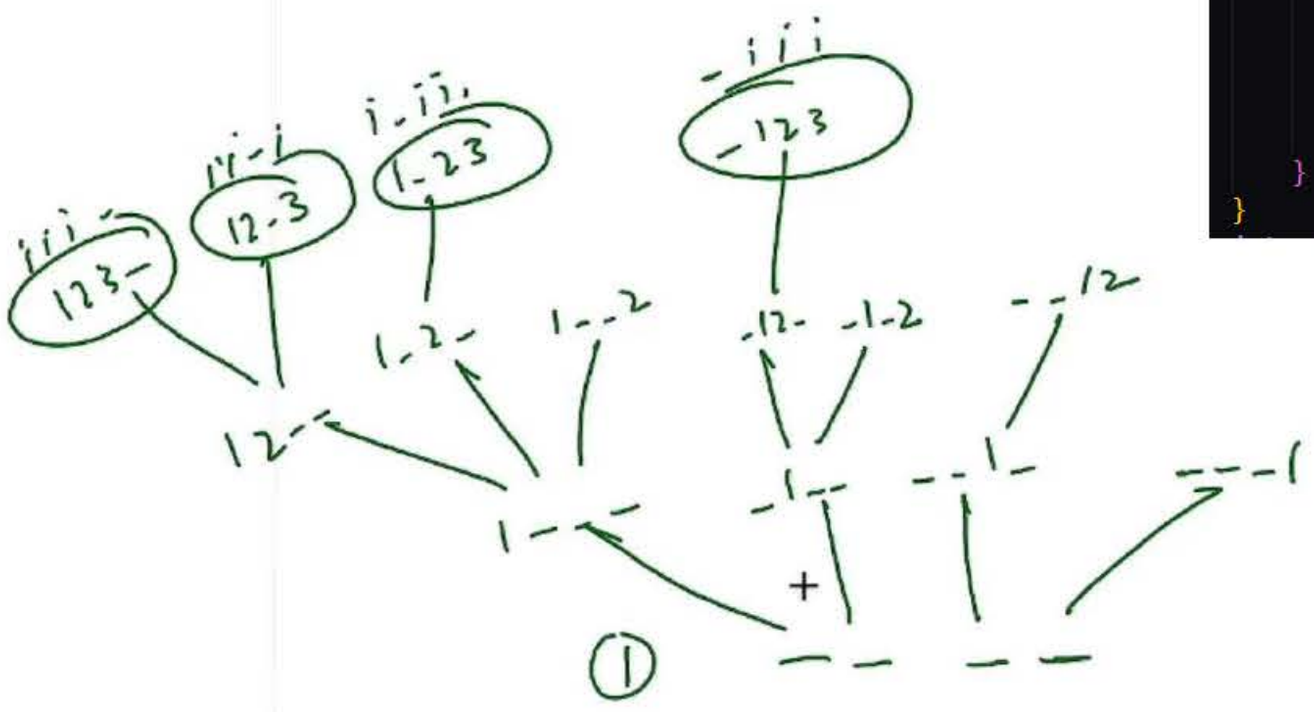




```
id permutation(string str, int boxNo, int totBox, int selItems, vector<int> usedItems){
  if(boxNo == totBox+1){
      if(selItems == usedItems.size())
          cout << str << endl;
      return;
  }
  //Konsa item dalna he
  for(int item = 0; item < usedItems.size(); item++){
      if(usedItems[item] == false){
          usedItems[item] = true;
          permutation(str + to_string(item+1), boxNo+1, totBox, selItems+1, usedItems);
          usedItems[item] = false;
      }
  }
//Item nahi dalna
permutation(str + "0", boxNo+1, totBox, selItems, usedItems);
```

COMBINATION 2

In combination
level - box
option - to place item or not
In combinations-2, we are using permutation to generate combination
ie nCr = nPr x 1/r!
level-items
option-in which box to place item (like permutation)
But here we can place current item only in boxes next to box in which prev item was placed(which prevents permutation among same boxes)



```
void combinations(vector<bool> boxes, int currItem, int ritems, int prevBox){
    if(currItem > ritems){
        for(bool i : boxes){
            if(i)
                cout << "i";
            else
                cout << "-";
        }
        cout << endl;
        return;
    }
    for(int box = prevBox+1; box < boxes.size(); box++){
        if(boxes[box] == false){
            boxes[box] = true;
            combinations(boxes, currItem+1, ritems, box);
            boxes[box] = false;
        }
    }
}
```

Permutation String

```
Sample Input
    aabb
Sample Output
    aabb
    abab
    abba
    baab
    baba
    bbaa
```

```
In this approach we are allowed to place the repeating character only in boxes
after the prev char was placed

void permutation(string str, int charNo, vector<char> boxes, int nonEmpty, unordered_map<char, int> mp){
                                                                                <char, prevIdx>
    if(charNo == str.size()){
        for(char &ch : boxes)
            cout << ch;
        cout << endl;
        return;
    }
    int boxNo;
    if(mp.find(str[charNo]) == mp.end())
        boxNo = 0;
    else
        boxNo = mp[str[charNo]];

    for(; boxNo < boxes.size(); boxNo++){
        if(boxes[boxNo] == '-'){
            boxes[boxNo] = str[charNo];
            mp[str[charNo]] = boxNo;
            permutation(str, charNo+1, boxes, nonEmpty+1, mp);
            mp.erase(str[charNo]);
            boxes[boxNo] = '-';
        }
    }
}
```

Using Permutation 1

```
We use permutation from combination techneque
Here there are n boxex and n items
ie n=r


So there is no option for the box to reject any item from placing into it
Only option is to place which item

ALso since there are duplicacies in input (ie aabb -> 2a's and 2b's), if we follow the standard PERMUTATION2 method
Duplicacy in final answer will be found as shown below

input: aabb

    b2    a---  a--- b---  b---
            \  \   |   /
            a\ a\  |b /b
               \ \ | /
    b1           ___   ___   ___  ___

To avoid that we can create a frequency map of each character so that there is options available for unique characters

    b2     a_ _ _ |a1b2    b _ _ _ | a2b1
                   \         /
                   a\       /b
                     \     /
    b1             __ __ __ __ | a2b2
```

```
void permutation(map<char, int> &FreqMap, int boxNo, int tbox, string asf){
    if(boxNo == tbox+1){
        cout << asf << endl;
        return;
    }
    for(auto &it : FreqMap){
        if(it.second > 0){
            it.second--;
            permutation(FreqMap, boxNo+1, tbox, asf + it.first);
            it.second++;
        }
    }
}
```

using permutation 2

LESSON LEARNT :

PERMUTATION 1 : TO AVOID DUPICACY WE PLACE THE DUPLICATE ITEM ONLY IN BOXES AFTER WHICH PREV DUPLICATE ITEM WAS PLACED

PERMUTATION 2 : TO AVOID DUPLICACY WE CREATE FREQUENCY MAP