Recursion :

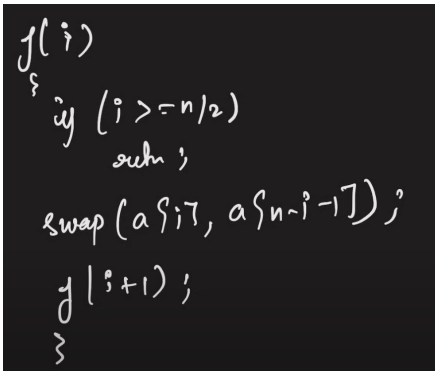
- When a function calls another one, it waits for it execute first before this executes further. It waits in memory stack.

Given a number N, print from 1 → N via backtracking

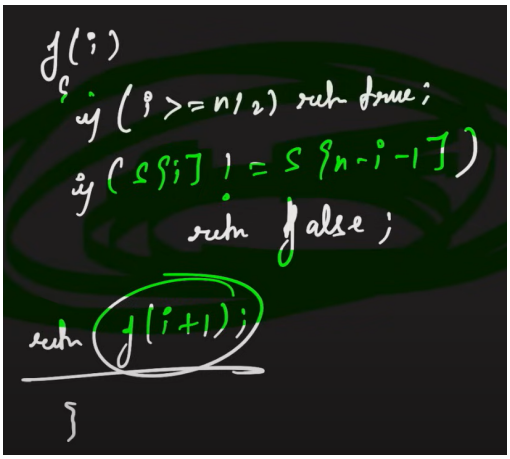
```
void f(int N) {
    if(N < 1) return;
    f(N-1);
    cout << N;
    // Doing something after function call is called backtracking
    // as we are doing this step when we try to move backward
}
```

How does reverse work internally in cpp ?

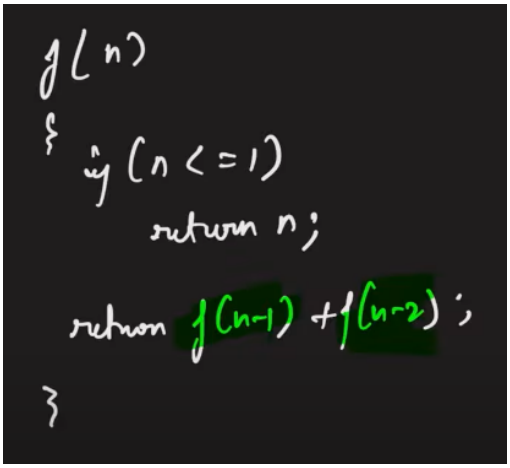
- The reverse function swaps elements starting from the beginning and end of the range, working its way towards the center.
- Or we can do till the starting n / 2 elements



Check if a given string is palindromic or not



- In Fibonacci series 0th element = 0, 1rst element = 1



Given a string print all of its subsequences

```
void solve(int i, string s, string &f) {
    if (i == s.length()) {
        cout << f << " ";
        return;
    }
    //picking
    f = f + s[i];
    solve(i + 1, s,  f);
    //poping out while backtracking
    f.pop_back();
    solve(i + 1, s,  f);
}
```

1. Print All possible answers : when ever base case satisfies keep adding answer

2. Print Only one answer : Change the return type of function to bool and check if we already got an answer before we do further recursion
3. Print Number of possible answers : return 1 if base case case satisfies, else 0. Then return the sum of all recursive functions.

Combination Sum 1

- array of distinct elements
- Each element can be taken more than once
- return number of distinct combinations (order doesn't matter)

⇒ As the elements are distinct, we need not sort it. It can be simply done with pick and stay & not pick and move ahead

---

Combination Sum 2

- Given an array where there can be duplicate elements
- Each element has to be taken only once
- return number of distinct combinations (order doesn't matter)

⇒ we can not simply do pick and move & not pick and move. As duplicity might occur

(1, 2, 1 == 1, 1, 2) order doesn't matter these are considered same

- Hence the first thing we will do is sort the array
- One method is to maintain a set to ensure no duplicates are added
- The other method is you will change the i'th element that you have picked for a particular combination so that it won't repeat.

```
class Solution {
    public:
    void findCombination(int ind, int target, vector<int> &arr, vector<vector<int>> &ans, vector<int>&ds) {
        if(target==0) {
            ans.push_back(ds);
            return;
        }
        for(int i = ind;i<arr.size();i++) {
            if(i>ind && arr[i]==arr[i-1]) continue;
            if(arr[i]>target) break;
            ds.push_back(arr[i]);
            findCombination(i+1, target - arr[i], arr, ans, ds);
            ds.pop_back();
        }
    }
public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(), candidates.end());
        vector<vector<int>> ans;
        vector<int> ds;
        findCombination(0, target, candidates, ans, ds);
        return ans;
    }
};
```

- inside the for loop of findCombination we allow the starting element to be pushed, later on when we come back we make sure we start the combination with a new element by simply continuing.

Subset Sums 1

- Given an array where there can be duplicate elements

- return sums of all possible subsets (need not be unique) but order matters here
- we can simply do it by pick and move & not pick and move.

Subset Sums 2

- Given an array where there can be duplicate elements
- return sums of all possible subsets (has to be unique) but order matters here
- Same logic as combination sum 2 problem

```
class Solution {
private:
    void findSubsets(int ind, vector<int> &nums, vector<int> &ds, vector<vector<int>>
        ans.push_back(ds);
        for(int i = ind;i<nums.size();i++) {
            if(i!=ind && nums[i] == nums[i-1]) continue;
            ds.push_back(nums[i]);
            findSubsets(i+1, nums, ds, ans);
            ds.pop_back();
        }
    }
public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        vector<vector<int>> ans;
        vector<int> ds;
        sort(nums.begin(), nums.end());
        findSubsets(0, nums, ds, ans);
        return ans;
    }
};
```

- **The main key difference between combination sum and subset sum is a target is given in CS but in SS we are not**

Given an array print all of its possible permutations :

```
class Solution {
  private:
    void recurPermute(vector < int > & ds,
    vector < int > & nums, vector < vector < int >> & ans, int freq[]) {
      if (ds.size() == nums.size()) {
        ans.push_back(ds);
        return;
      }
      for (int i = 0; i < nums.size(); i++) {
        if (!freq[i]) {
          ds.push_back(nums[i]);
          freq[i] = 1;
          recurPermute(ds, nums, ans, freq);
          freq[i] = 0;
          ds.pop_back(); // backtracking
        }
      }
    }
  public:
    vector < vector < int >> permute(vector < int > & nums) {
```

```
        vector < vector < int >> ans;
        vector < int > ds;
        int freq[nums.size()];
        for (int i = 0; i < nums.size(); i++) freq[i] = 0;
        recurPermute(ds, nums, ans, freq);
        return ans;
    }
};
```

OR

```
class Solution {
private:
    void recurPermute(int index, vector<int> &nums, vector<vector<int>> &ans) {
        if(index == nums.size()) {
            ans.push_back(nums);
            return;
        }
        for(int i = index;i<nums.size();i++) {
            swap(nums[index], nums[i]);
            recurPermute(index+1, nums, ans);
            swap(nums[index], nums[i]);
        }

    }
public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> ans;
        recurPermute(0, nums, ans);
        return ans;
    }
};
```

- **BackTracking is a Brute-force method : Used when the search space is small !**

N Queen

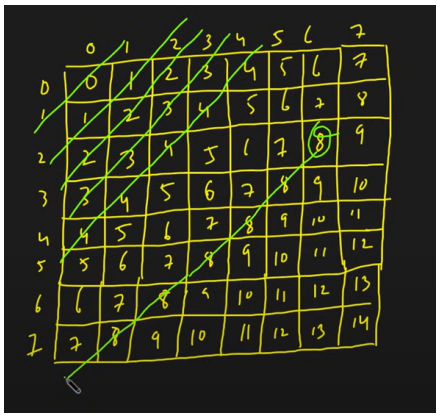
```
class Solution {
  public:
    bool isSafe1(int row, int col, vector < string > board, int n) {

      int duprow = row;
      int dupcol = col;

            // check upper diagonal
      while (row >= 0 && col >= 0) {
        if (board[row][col] == 'Q')
          return false;
        row--;
        col--;
      }

      col = dupcol;
      row = duprow;
```

```
      // check left
      while (col >= 0) {
        if (board[row][col] == 'Q')
          return false;
        col--;
      }

            // check lower diagonal
      row = duprow;
      col = dupcol;
      while (row < n && col >= 0) {
        if (board[row][col] == 'Q')
          return false;
        row++;
        col--;
      }

      // we need not check the remaining directions
      return true;
    }

  public:
    void solve(int col, vector < string > & board,
    vector < vector < string >> & ans, int n) {
      if (col == n) {
        ans.push_back(board);
        return;
      }
      for (int row = 0; row < n; row++) {
        if (isSafe1(row, col, board, n)) {
          board[row][col] = 'Q';
          solve(col + 1, board, ans, n);
          board[row][col] = '.';
        }
      }
    }

  public:
    vector < vector < string >> solveNQueens(int n) {
      vector < vector < string >> ans;
      vector < string > board(n);
      string s(n, '.');
      for (int i = 0; i < n; i++) {
        board[i] = s;
      }
      solve(0, board, ans, n);
      return ans;
```

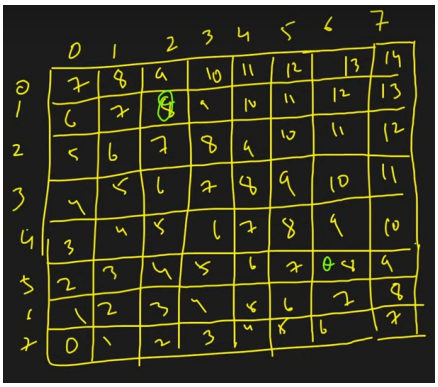
```
    }
};
```

isSafe is completely omitted using hash table :

```
class Solution {
  public:
    void solve(int col, vector < string > & board,
    vector < vector < string >> & ans, vector < int > & leftrow, vector < int > & upp
      if (col == n) {
        ans.push_back(board);
        return;
      }
      for (int row = 0; row < n; row++) {
        if (leftrow[row] == 0 && lowerDiagonal[row + col] == 0 &&
        upperDiagonal[n - 1 + col - row] == 0) {
          board[row][col] = 'Q';
          leftrow[row] = 1;
          lowerDiagonal[row + col] = 1;
          upperDiagonal[n - 1 + col - row] = 1;
          solve(col + 1, board, ans, leftrow, upperDiagonal, lowerDiagonal, n);
          board[row][col] = '.';
          leftrow[row] = 0;
          lowerDiagonal[row + col] = 0;
          upperDiagonal[n - 1 + col - row] = 0;
        }
      }
    }

  public:
    vector < vector < string >> solveNQueens(int n) {
      vector < vector < string >> ans;
      vector < string > board(n);
      string s(n, '.');
      for (int i = 0; i < n; i++) {
        board[i] = s;
      }
      vector < int > leftrow(n, 0), upperDiagonal(2 * n - 1, 0),
          lowerDiagonal(2 * n - 1, 0);
      solve(0, board, ans, leftrow, upperDiagonal, lowerDiagonal, n);
      return ans;
    }
};
```



lower diagonal



Upper diagonal

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy **all of the following rules**:

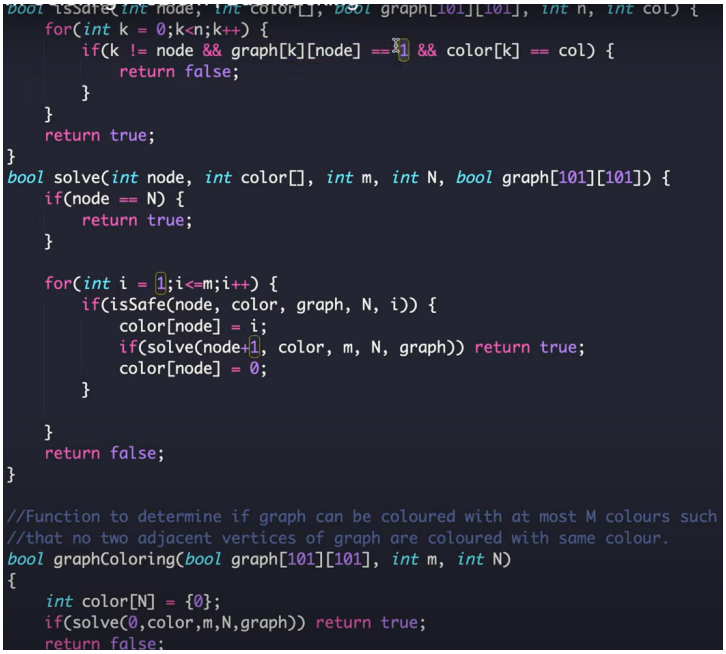
1. Each of the digits `1-9` must occur exactly once in each row.
2. Each of the digits `1-9` must occur exactly once in each column.
3. Each of the digits `1-9` must occur exactly once in each of the 9 `3x3` sub-boxes of the grid.

The `'.'` character indicates empty cells.

```
bool isValid(vector < vector < char >> & board, int row, int col, char c) {
  for (int i = 0; i < 9; i++) {
    if (board[i][col] == c)
      return false;

    if (board[row][i] == c)
      return false;

    if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c) //**
      return false;
  }
  return true;
}

bool solveSudoku(vector < vector < char >> & board) {
  for (int i = 0; i < board.size(); i++) {
    for (int j = 0; j < board[0].size(); j++) {
      if (board[i][j] == '.') {
        for (char c = '1'; c <= '9'; c++) {
          if (isValid(board, i, j, c)) {
            board[i][j] = c;

            if (solveSudoku(board))
```

```
              return true;
            else
              board[i][j] = '.';
          }
        }

        return false;
      }
    }
  }
  return true;
}
```

Coloring an undirected graph, with at-max m colors such that no adjacent node is of same color



```
bool isSafe(int node, int color[], bool graph[101][101], int n, int col) {
    for(int k = 0;k<n;k++) {
        if(k != node && graph[k][node] ==1 && color[k] == col) {
            return false;
        }
    }
    return true;
}
bool solve(int node, int color[], int m, int N, bool graph[101][101]) {
    if(node == N) {
        return true;
    }

    for(int i = 1;i<=m;i++) {
        if(isSafe(node, color, graph, N, i)) {
            color[node] = i;
            if(solve(node+1, color, m, N, graph)) return true;
            color[node] = 0;
        }

    }
    return false;
}

//Function to determine if graph can be coloured with at most M colours such
//that no two adjacent vertices of graph are coloured with same colour.
bool graphColoring(bool graph[101][101], int m, int N)
{
    int color[N] = {0};
    if(solve(0,color,m,N,graph)) return true;
    return false;
```

Given a String return all possible partitions of the string such that the resultant substring are palindromic in nature

```cpp
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string> > res;
        vector<string> path;
        func(0, s, path, res);
        return res;
    }

    void func(int index, string s, vector<string> &path,
              vector<vector<string> > &res) {
        if(index == s.size()) {
            res.push_back(path);
            return;
        }
        for(int i = index; i < s.size(); ++i) {
            if(isPalindrome(s, index, i)) {
                path.push_back(s.substr(index, i - index + 1));
                func(i+1, s, path, res);
                path.pop_back();
            }
        }
    }

    bool isPalindrome(string s, int start, int end) {
        while(start <= end) {
            if(s[start++] != s[end--])
                return false;
        }
        return true;
    }
};
```

Rat in maze : start from {0, 0} → {n-1, n-1}. Can move in 4 directions.

```cpp
class Solution {
  void findPathHelper(int i, int j, vector < vector < int >> & a,
  int n, vector < string > & ans, string move,
    vector < vector < int >> & vis) {
    if (i == n - 1 && j == n - 1) {
      ans.push_back(move);
      return;
    }

    // downward
    if (i + 1 < n && !vis[i + 1][j] && a[i + 1][j] == 1) {
      vis[i][j] = 1;
      findPathHelper(i + 1, j, a, n, ans, move + 'D', vis);
      vis[i][j] = 0;
    }

    // left
    if (j - 1 >= 0 && !vis[i][j - 1] && a[i][j - 1] == 1) {
      vis[i][j] = 1;
      findPathHelper(i, j - 1, a, n, ans, move + 'L', vis);
      vis[i][j] = 0;
    }
```

```
    // right
    if (j + 1 < n && !vis[i][j + 1] && a[i][j + 1] == 1) {
      vis[i][j] = 1;
      findPathHelper(i, j + 1, a, n, ans, move + 'R', vis);
      vis[i][j] = 0;
    }

    // upward
    if (i - 1 >= 0 && !vis[i - 1][j] && a[i - 1][j] == 1) {
      vis[i][j] = 1;
      findPathHelper(i - 1, j, a, n, ans, move + 'U', vis);
      vis[i][j] = 0;
    }

  }
  public:
    vector < string > findPath(vector < vector < int >> & m, int n) {
      vector < string > ans;
      vector < vector < int >> vis(n, vector < int > (n, 0));

      if (m[0][0] == 1) findPathHelper(0, 0, m, n, ans, "", vis);
      return ans;
    }
};
```

Given an integer n. return Kth permutation of first n natural numbers

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