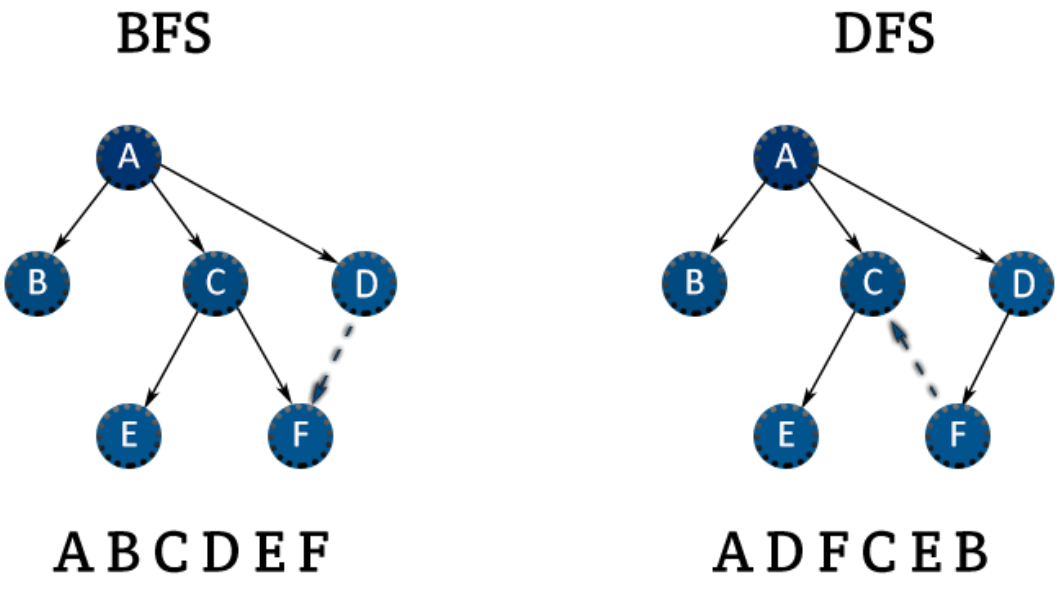
Overview of important Algorithms

- Searching
  - Binary Search
  - Depth First Search for trees and graphs
    - start from the top of a tree and go as deep as possible along the same branch
    - once you are at the bottom then go to nearest unvisited node usually a sibling of the deepest node
      - this process is called **Backtracking**
    - used to solve a maze
    - `O(number of nodes + number of branches)`
  - Breadth First Search for trees and graphs
    - you don't go to deepest point like DFS
    - instead you make sure that the sibling node has been visited
    - once you are on a node look at its children and add them to a queue and then you visit the node in the queue and add them to visited array and remove them from sibling queue
    - if the node in the queues has more children then add them to queue when marking it visited
    - used in chess
    - `O(number of nodes + number of branches)`



- Sorting
  - Insertion Sort

    - compares the nth element with (n+1)th element and swaps them if nth element is larger
    - best case `O(n)` if everything is already sorted
    - worst case `O(n^2)` when nothing is sorted beforehand
- Merge Sort
    - divide and conquer and conquer by divide and conquer and so on
    - recursion is used
    - splits array in half till we have pairs of 2
    - then all pairs of 2 are sorted and then 2 pairs of 2 are merged and sorted till the array is completely merged back again
    - best and worst case are same `O(n log n)`
- Quick Sort
    - recursive like merge sort so divides and conquers
    - we choose a pivot element of the array which is closest to the median of the array elements
    - then we split the lists into 2 such that one list has elements smaller than the pivot element and one where all elements are greater than the pivot element
    - we repeat the same on these 2 lists
    - we move the pivot element to the end of the list
    - we place 2 pointers one on the 0th index and the 2nd on the 2nd last element and compare the two if the 0th one is larger we swap
    - deep doing it till the 2 pointers meet
    - when they meet replace that element with the last one
    - we know have 2 lists like we wanted and we can do the same thing on them individually
    - best case `O(n log n)`
    - worst case `O(n^2)`
    - still can be 2 to 3 times faster than merge sort by reducing the chances of worst case
    - needs less memory `O(log n)` than merge sort `O(n)`
- Greedy Algorithm
    - it makes the best possible decision at every local step
    - when not to be greedy
        - not meant for efficiency
    - when to be greedy

- when you don't want to find the most efficient way out of millions of permutations then greedy might be a good enough solution
- when optimal solution not possible and brute force is not acceptable become greedy

Recursion

- a recursive function should have a terminating condition also called as a base condition

    - the values in the scope of the function can be used before(ascending) or after(descending) the termination condition and recursive call

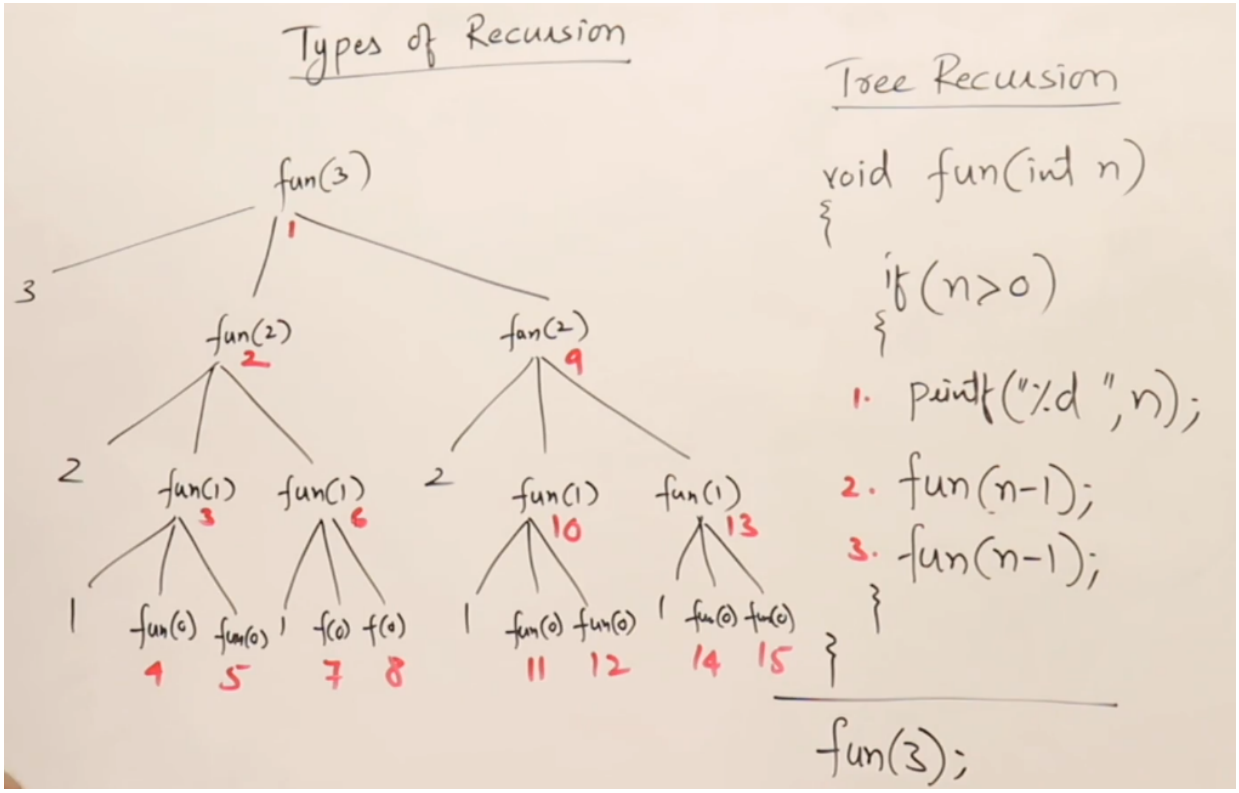
```cpp
#include <iostream>
using namespace std;

void head(int n)
{
    if (n > 0)
    {
        head(n - 1);
        cout << n << " ";
    }
}

void tail(int n)
{
    if (n > 0)
    {
        cout << n << " ";
        tail(n - 1);
    }
}

int main()
{
    head(10);
    cout << endl;
    tail(10);

    return 0;
```

```
}
// 1 2 3 4 5 6 7 8 9 10
// 10 9 8 7 6 5 4 3 2 1
```

- use static variables in recursive function when you need a counter and don't want the counter to reset on every recursive call
  - static variable will have a single copy for all recursive calls and will not be a local variable of the scope of a recursive function
  - it is like global but more restricted
- types of recursion
  - tail
    - when the function calls itself in the last line of the function
    - easier to convert recursive logic to iterative
  - head
    - when the function calls itself in the first line of the function
    - harder to convert recursive logic to iterative
  - tree
    - opposite of tree recursion is linear recursion when the recursive function calls itself only one time
    - in tree recursion the recursive function calls itself more than one times

```cpp
#include <iostream>
using namespace std;
void tree(int n)
{
    if (n > 0)
    {
        cout << n << " ";
        tree(n - 1);
        tree(n - 1);
    }
}
```

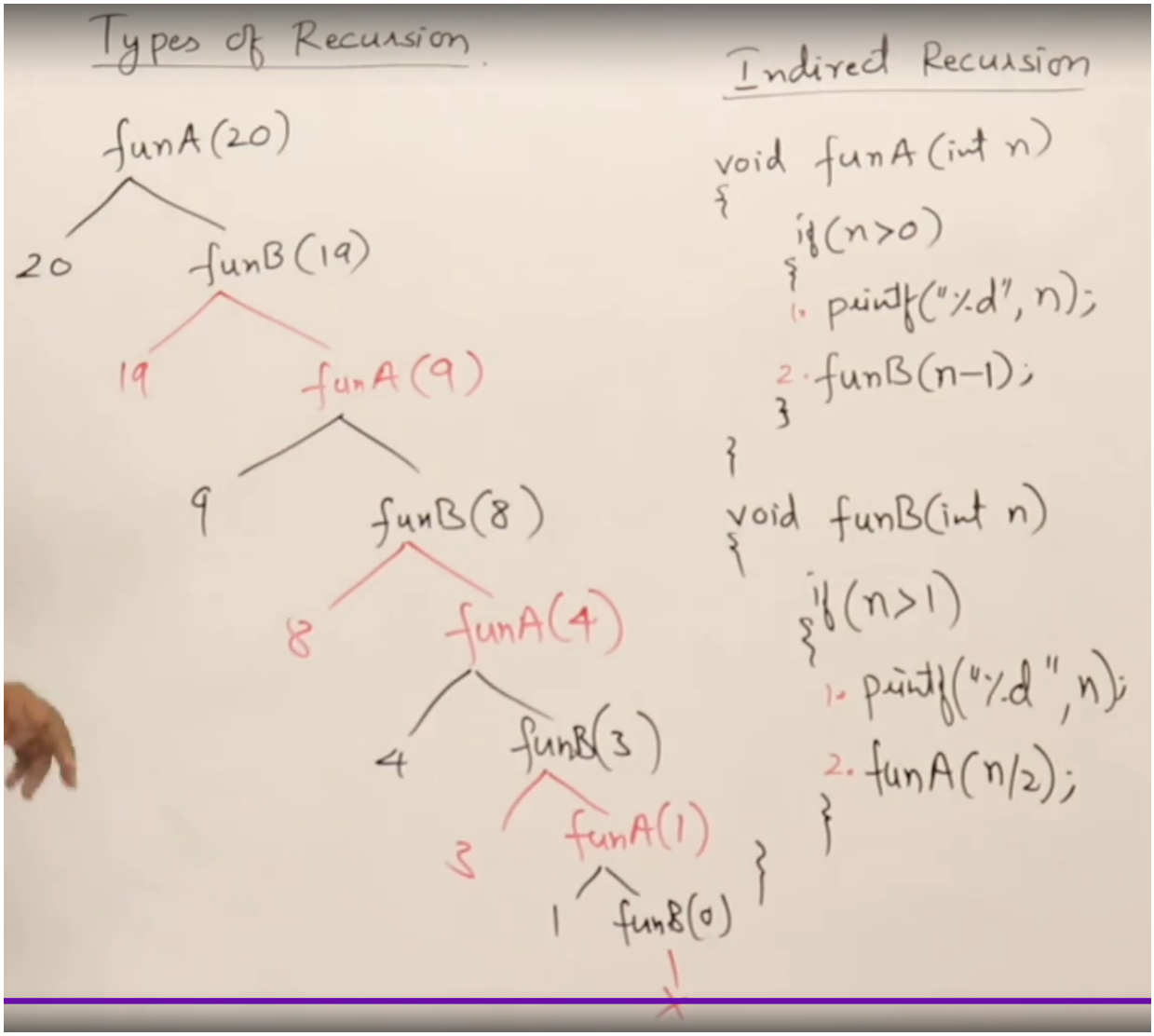
```
int main()
{
    tree(3);
    return 0;
}
// 3 2 1 1 2 1 1
// Time O(2^n)
// Space O(n)
```

- 

- indirect
  - when a function A calls B and B calls C and C calls A

```
#include <iostream>
using namespace std;
void funB(int n);

void funA(int n)
{
    if (n > 0)
    {
        cout << n << " ";
        funB(n - 1);
    }
}
void funB(int n)
```

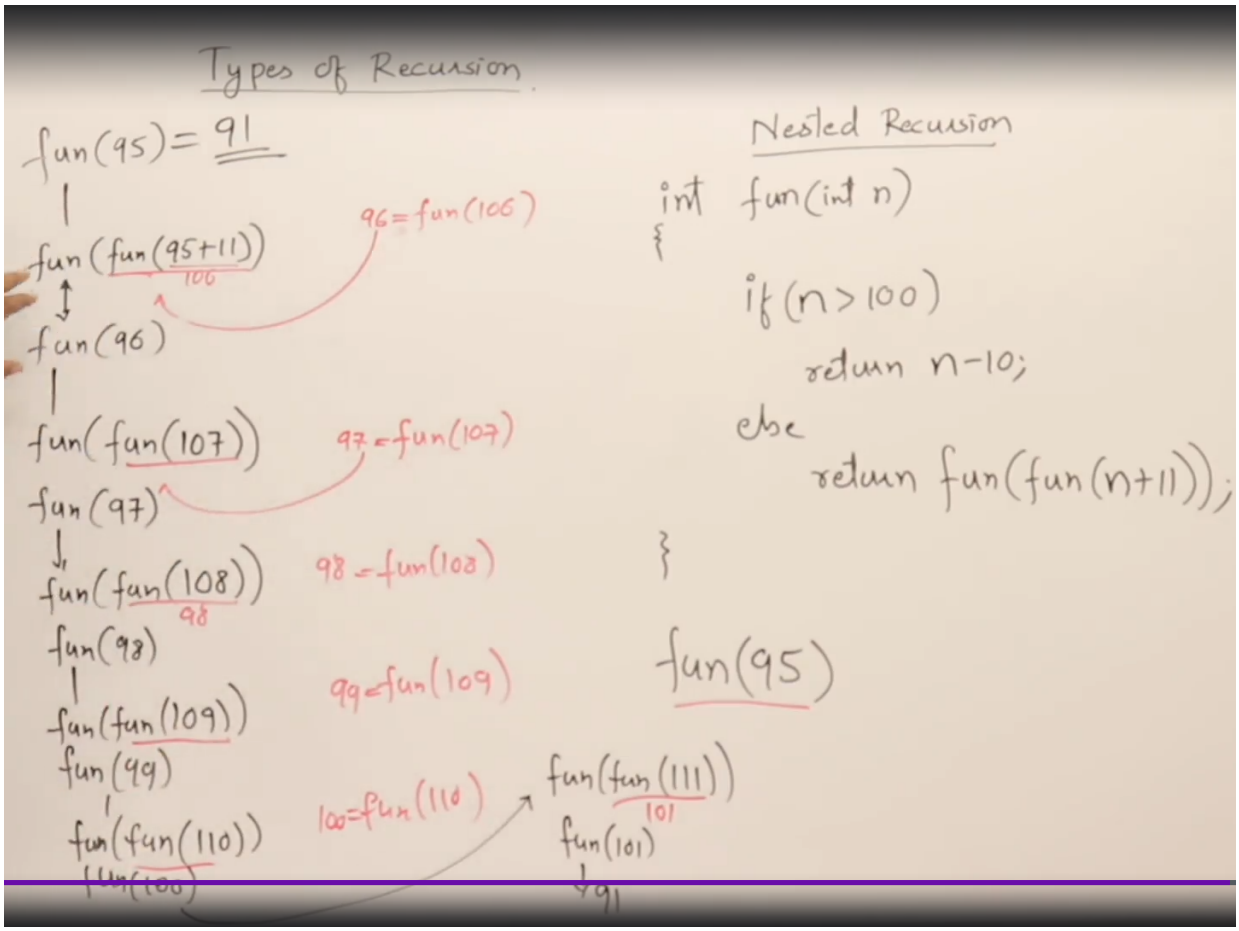
```
{
    if (n > 1)
    {
        cout << "\n"
             << n << " ";
        funA(n / 2);
    }
}

int main()
{
    funA(20);
    return 0;
}
// 20
// 19 9
// 8 4
// 3 1
```

- 

- nested

- parameter of a recursive call function is the same function

```
#include <iostream>
using namespace std;
int fun(int n)
{
    if (n > 100)
        return n - 10;
    return fun(fun(n + 11));
}
int main()
{
    cout << fun(95); // 91
        return 0;
}
```

- 

- Implementing pow function from cmath using recursion

```
#include <iostream>
using namespace std;
int pow(int k, int p) { return p == 0 ? 1 : pow(k, p - 1) * k; }
int main()
{
    cout << "Enter constant and power: ";
```

```
    int con, pwr;
    cin >> con >> pwr;
    cout << con << "^" << pwr << " = " << pow(con, pwr);
    return 0;
}
```

- optimization for 2^8 instead of multiplying 2 8 times shouldn't we half and square like (2^2)^4 = 4^4
  - this way we can reduce the stack height and increase memory efficiency
  - so if the power is even we half it and then we square the constant
  - else if the power is odd like 2^9 we can still do 2 x 2^8 and so on

```
#include <iostream>
using namespace std;
int pow(int k, int p) { return p == 0 ? 1 : p % 2 == 0 ? pow(k
* k, p / 2)
                                                        : k *
pow(k * k, (p - 1) / 2); }
int main()
{
    cout << "Enter constant and power: ";
    int con, pwr;
    cin >> con >> pwr;
    cout << con << "^" << pwr << " = " << pow(con, pwr);
    return 0;
}
```

- Taylor Series using recursion is a combination of sum till n, power, factorial using recursion
  - to print `e^x = 1 + x/1 + x^2/2! + x^3/3! + x^4/4! + ... till n terms`
  - we need to use static variables as 3 variables are involved but we can return only one
    - the program will be less efficient if we don't use power and factorial as static variables as we will have to calculate the complete factorial over and over again
      - if factorial would have been static we just need to multiply a new number with the factorial of the previous number as n! = n x

(n-1)!
    - similarly we have to find x^n every time but if static we can store x^(n-1) and multiply x once

```
#include <iostream>
using namespace std;
float e(int x, int n)
{
    if (n == 0)
        return 1;

    static float pwr = 1, fac = 1;
    float res = e(x, n - 1);
    pwr *= x;
    fac *= n;
    return res + pwr / fac;
}
int main()
{
    cout << "Enter x and n: ";
    int x, n;
    cin >> x >> n;
    cout << "e^" << x << " till n precision is " << e(x, n);
    return 0;
}
```

- optimizing using Horner's Rule
    - earlier the number of times we were multiplying was `O(n^2)` but using Horner's Rule it can be `O(1)`
    - to print `e^x = 1 + x/1(1 + x/2(1 + x/3(1 + x/4 + ... till n terms)))`
        - we keep taking commons out and this reduces number of multiplications that are needed to be performed
        - we find the value for the innermost bracket lets say (1+ x/4) here and multiply it with the common multiple x/3 and add 1 to it and go on recursively
        - using iteration

```
#include <iostream>
using namespace std;
float e(int x, int n)
```

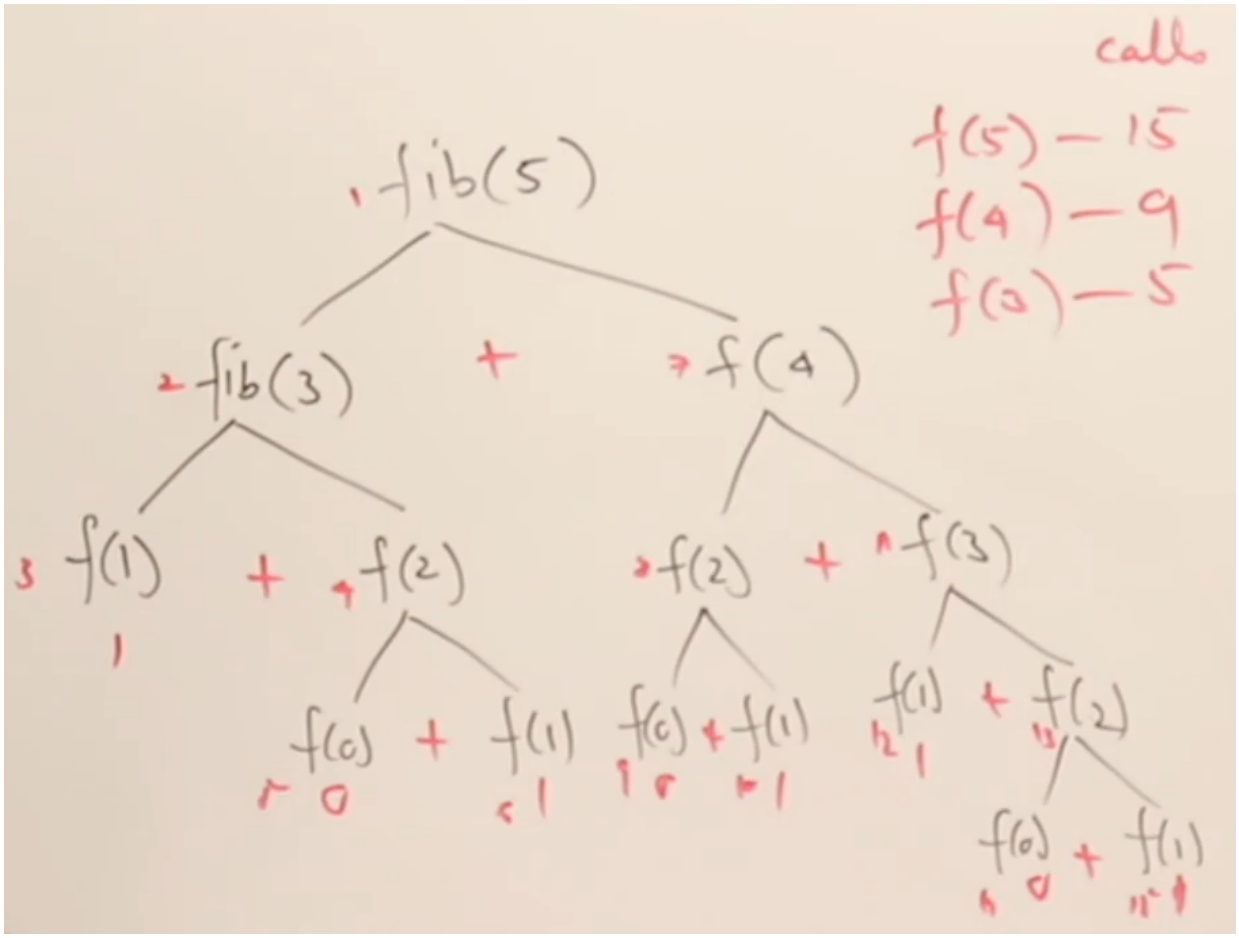
```
{
    int res = 1;
    for (; n > 0; n--)
        res = 1 + x * res / n;
    return res;
}
int main()
{
    cout << "Enter x and n: ";
    int x, n;
    cin >> x >> n;
    cout << "e^" << x << " till n precision is " <<
e(x, n);
    return 0;
}
```

- using recursion

```
#include <iostream>
using namespace std;
float e(int x, int n)
{
    static int res = 1;
    if (n == 0)
        return res;
    res = 1 + x * res / n;
    return e(x, n - 1);
}
int main()
{
    cout << "Enter x and n: ";
    int x, n;
    cin >> x >> n;
    cout << "e^" << x << " till n precision is " <<
e(x, n);
    return 0;
}
```

- Fibonacci Series
  - using recursion `O(2^n)`

```cpp
#include <iostream>
using namespace std;
int fibo(int n)
{
    if (n <= 1)
        return n;
    return fibo(n - 2) + fibo(n - 1); // as the function calls itself 2 times with n as arg so O(2^n)
}
```

- 

- here we can see that fib(3) and fib(2) get calculated over and over as the value is not stored
  - it is a case of excessive recursion and we can fix it by using static variables
    - we create a static array that stores the fib(n) at index n and the default values for all the elements is -1 so we can check do we need to find fib(n) at every step so `O(n)`
    - this process is called memoization

```cpp
#include <bits/stdc++.h>
using namespace std;
int fibo(int n)
{
```

```
    static vector<int> memo(n + 1, -1);
    if (n <= 1)
    {
        memo[n] = n;
        return n;
    }
    else if (memo[n - 2] == -1)
        memo[n - 2] = fibo(n - 2);
    if (memo[n - 1] == -1)
        memo[n - 1] = fibo(n - 1);
    return memo[n - 2] + memo[n - 1];
}
```

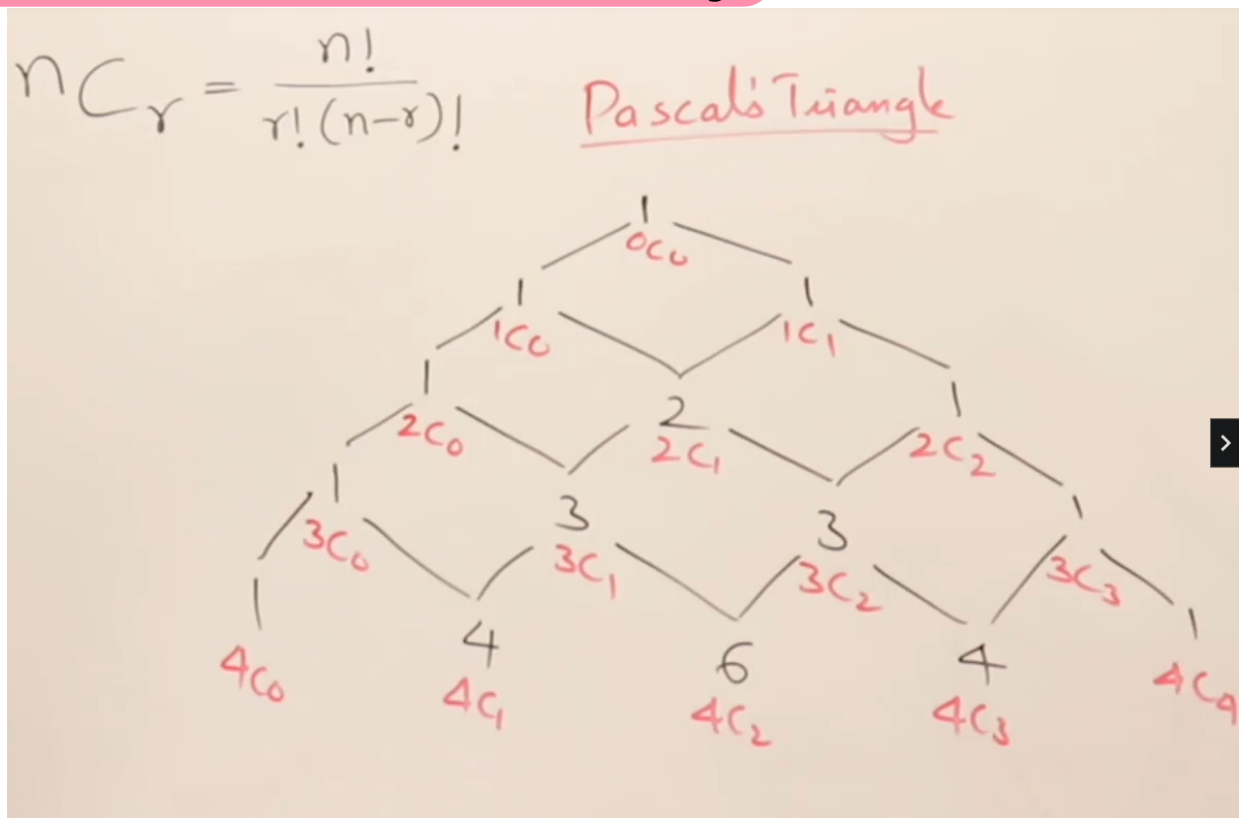
- using iteration `O(n)`

```
#include <iostream>
using namespace std;
int fibo(int n)
{
    if (n <= 1)
        return n;
    int t0 = 0, t1 = 1, s = 0;
    for (int i = 2; i <= n; i++)
    {
        s = t0 + t1;
        t0 = t1;
        t1 = s;
    }
    return s;
}
```

- nCr = n!/(r!*(n-r)!)
    - using iteration

```
#include <iostream>
using namespace std;
int fac(int n)
{
    int res = 1;
    for (int i = 1; i <= n; i++)
        res *= i;
    return res;
```

```
}
int main()
{
    cout << "Enter n and r: ";
    int n, r;
    cin >> n >> r;
    if (r > n)
    {
        cout << "Invalid Input " << r << " grtr than " << n;
        return -1;
    }
    cout << n << "C" << r << " = " << fac(n) / (fac(r) * fac(n
- r));
    return 0;
}
```

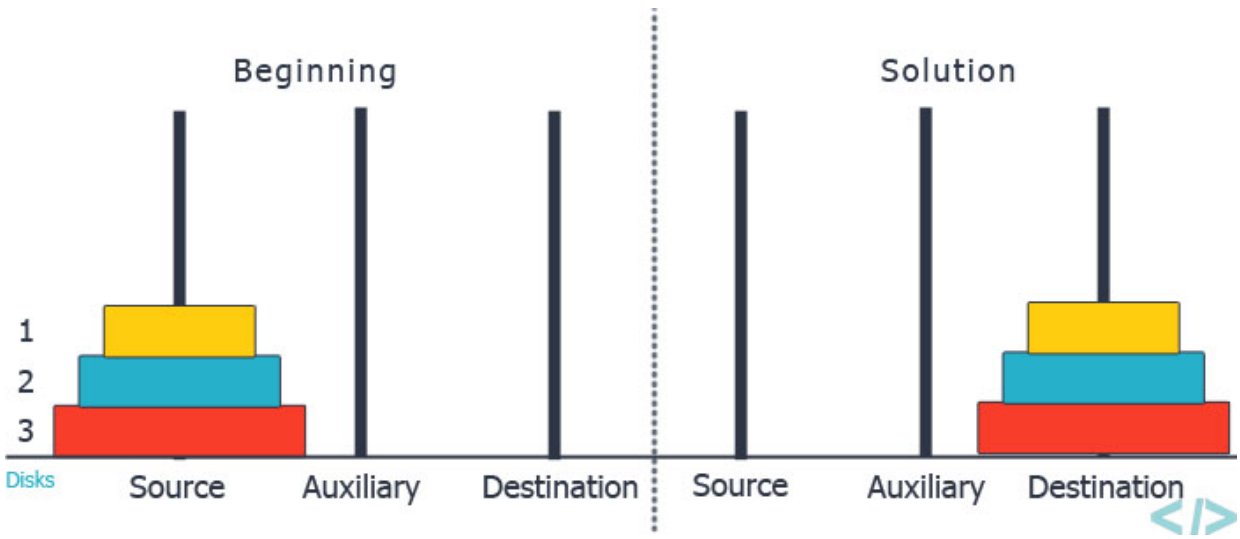
- for recursion we need to use Pascal's Triangle
- 

    - here we have rows and cols and an element having both `\` and `/` points to elements whose sum is equal to itself
        - observe how 2C1 points to above elements 1C0 and 1C1 and 2C1 = 2 = 1C0 + 1C1 = 1 + 1
        - so we can say 6 is 4C2 is obtained from 3+3 of 3C1 and 3C2 which themselves are obtained from a sum
        - our base condition can be outlined by observing that the extreme leftmost and rightmost elements are always 1 and the topmost element is also always 1

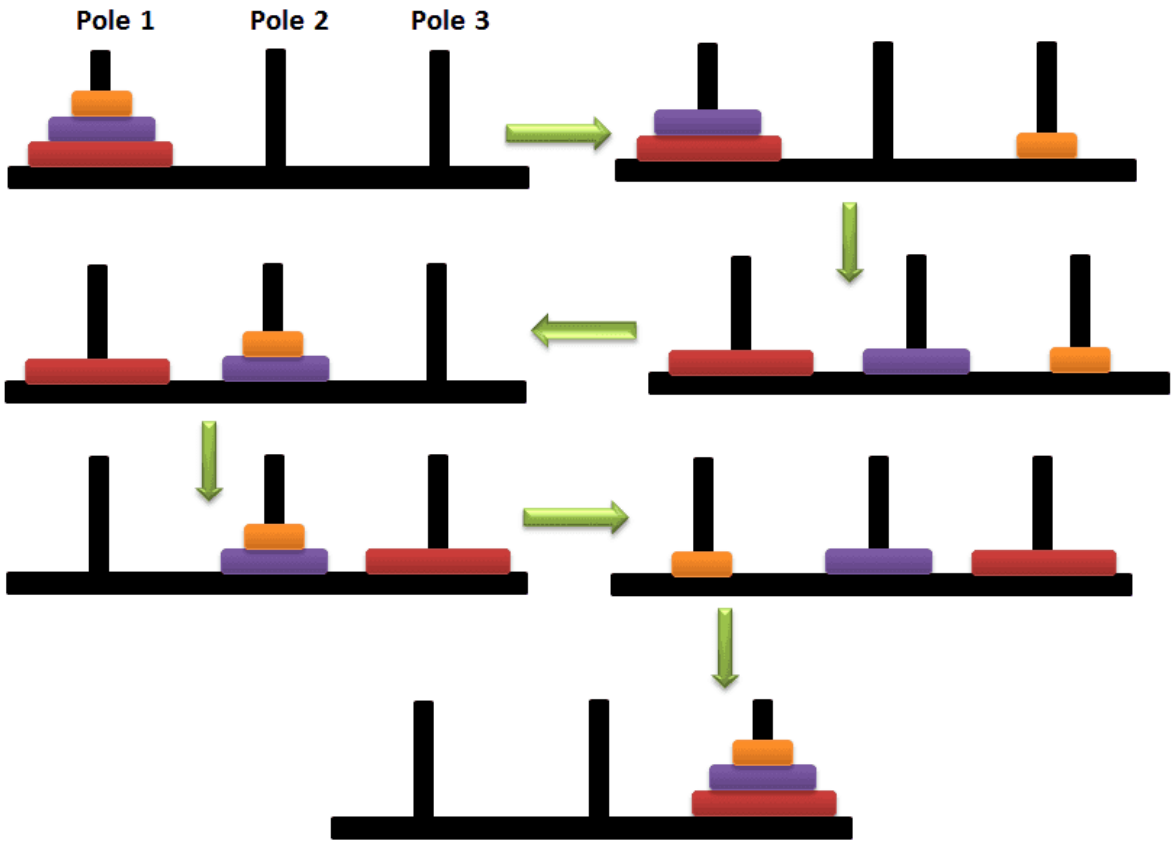
- so `nCr = (n-1)C(r-1) + (n-1)Cr`
- we go bottom to up for recursion and then come down from top to bottom when returning

```cpp
#include <iostream>
using namespace std;
int C(int n, int r)
{
    if (r == 0 || n == r)
        return 1; // extreme left and right base condition
    return C(n - 1, r - 1) + C(n - 1, r);
}
int main()
{
    cout << "Enter n and r: ";
    int n, r;
    cin >> n >> r;
    if (r > n)
    {
        cout << "Invalid Input " << r << " grtr than " << n;
        return -1;
    }
    cout << n << "C" << r << " = " << C(n, r);
    return 0;
}
```

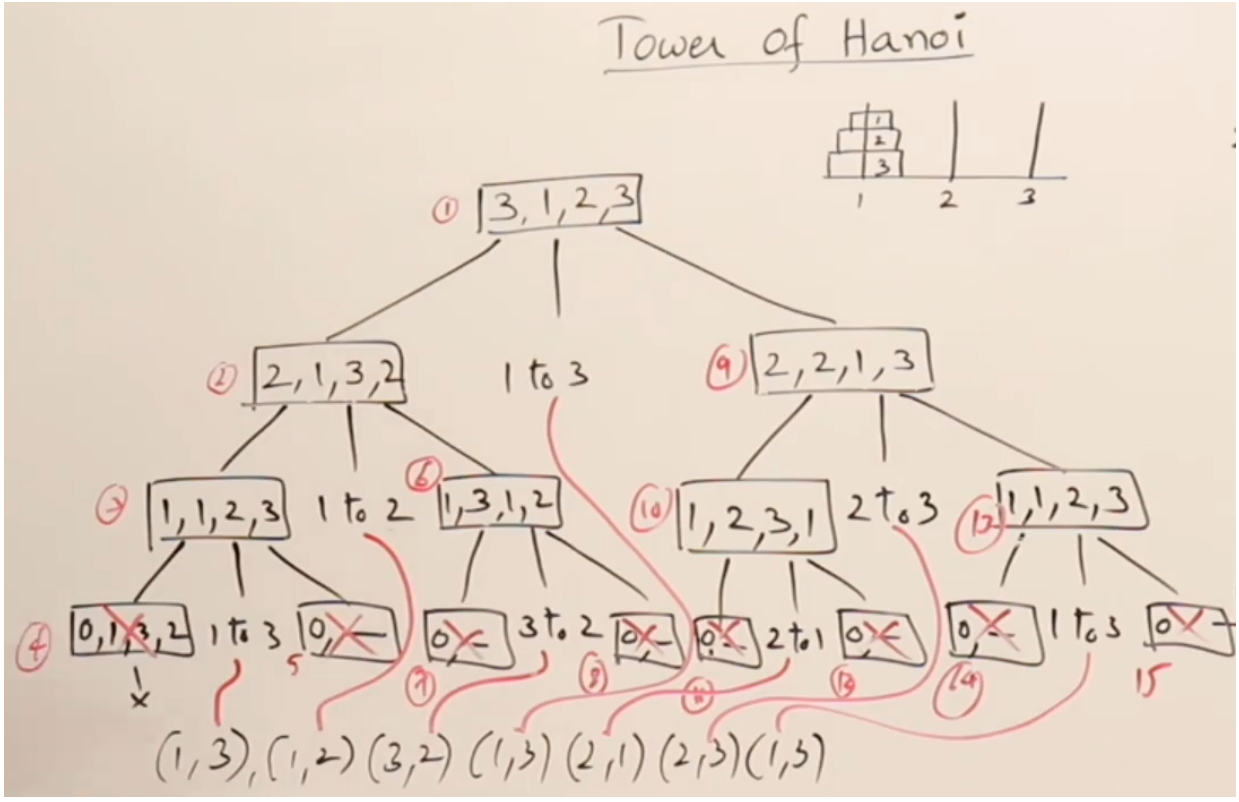
- Tower of Hanoi
  - Question:



    - Auxiliary pole is for helping like temp variable for swapping
        - discs are always sorted in the source pole
- Procedure to Solution: `O(2^n)`

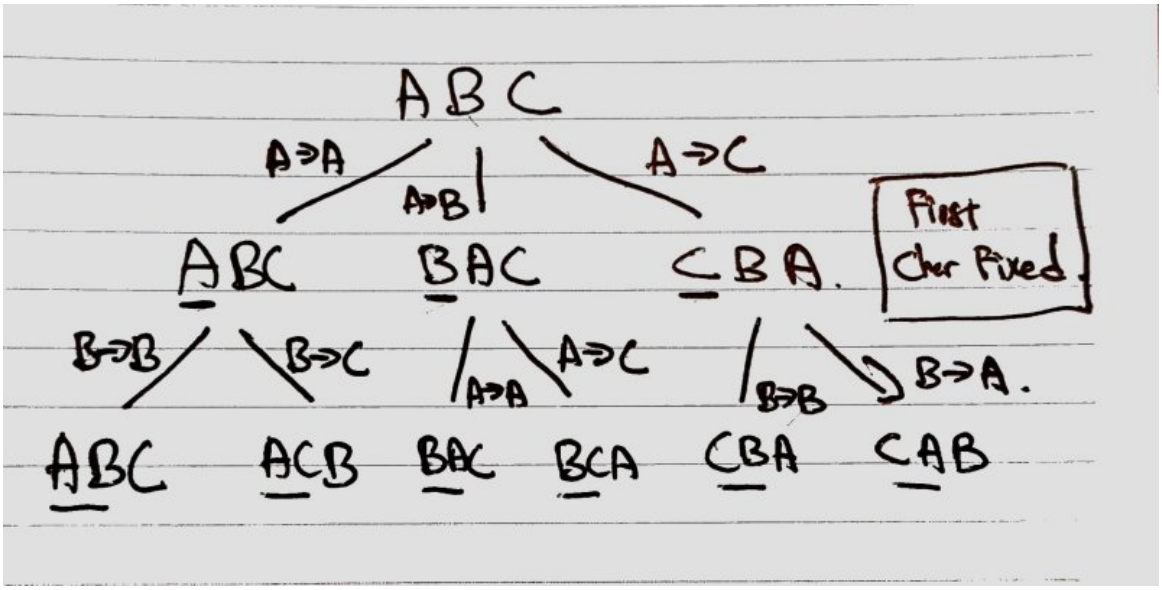


- cases
    - if the number of discs is one then just move it from pole 1 to 3
    - if the number of discs are 2 then
        1. move the smaller disc to middle pole
        2. move the larger disc from 1st to the 3rd pole
        3. move the smallest disc from 2nd to 3rd pole
    - if number of discs are 3 then
        1. ignore the largest disc (3rd disc) and move the 2 discs to 2nd pole as if the number of discs were 2
        2. move the largest disc to 3rd pole
        3. move the 2 discs in 2nd pole to the 3rd pole as if the number of discs were 2
    - if number of discs are n then
        1. ignore the largest disc (nth disc) and move the n-1 discs to 2nd pole as if the number of discs were n and do this recursively
        2. move the largest disc (nth disc) to 3rd pole
        3. move the n-1 discs in 2nd pole to the 3rd pole as if the number of discs were n

- 


```
#include <iostream>
using namespace std;

void toh(int n, int sp1, int mp2, int dp3)
// source pole 1, middle pole 2 and destination pole 3 are just
pole numbers and n discs are stored in sp1 and we want to move it
to dp3 in a specific order
{
    if (n < 0)
        return;
    static int ctr = 0;
    toh(n - 1, sp1, dp3, mp2); // move (n-1)th disc from sp1 to
mp2 using dp3 as Auxiliary Pole
    ctr++;
    cout << "[Step " << ctr << "] move disc from " << sp1 << " to
" << dp3 << endl; // print what was done in the above step
    toh(n - 1, mp2, sp1, dp3);
// move (n-1)th disc from mp2 to dp3 using sp1 as Auxiliary Pole
    // when n=1 the topmost pole is moved and then n=2 so the one
below it is moved and so on
}

int main()
{
    toh(2, 1, 2, 3); // means sp1 is the 1st pole, mp2 is the 2nd
pole and dp3 is the 3rd pole
    return 0;
```

```
}

/**
[Step 1] move disc from 1 to 3
[Step 2] move disc from 1 to 2
[Step 3] move disc from 3 to 2
[Step 4] move disc from 1 to 3
[Step 5] move disc from 2 to 1
[Step 6] move disc from 2 to 3
[Step 7] move disc from 1 to 3
*/
```

- Permutations of a String without swapping
  - number of possible permutations = (length of the string)!
  - imagine a tree which has the original string as the root node
    - the number of children to this root node = length of the string and each child will be an element of the string(underlined in 2nd row)
      - these children will each be a parent to length - 1 nodes(the ones that are not underlined in the 3rd row)
      - 

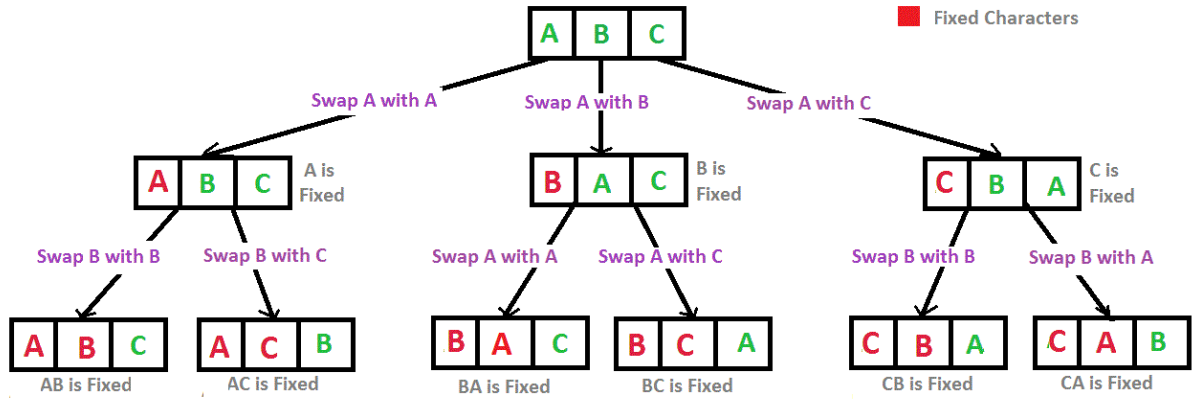
        - a tree whose leaf nodes are the solution to the program is called a state space tree
  - we need a static flag array so that we can find out what elements of the string can be children as child and parent cannot be the same element
    - i is the counter variable for this array initially pointing to the 0th index
  - we also need a static result array to store the permutations of the original string

    - k is the counter variable for this array initially pointing to the 0th index
    - when we have used all the characters from the original string then the all flag array elements will be set to true then we can add the `\0` at the end and print it
    - so adding any element at index n of the original array to the result array leads to toggling of nth index of flag array to true meaning that the index or original array is already used once in the resultant array
- check flag array for the first occurrence of `result[i]` is false
    - when found at i set `result[k] = original[i]`
- recursive call with k+1 and i should automatically reset to 0
- backtracking is used as it is depth first algorithm
    - so below the recursive call reset the flag array at index i back to false

```
#include <iostream>
using namespace std;
void permute(char strray[], int k = 0)
{
    static bool flag[10];
    static char result[10];
    if (strray[k] == '\0')
    {
        result[k] = '\0';
        cout << result << endl;
        return;
    }
    for (int i = 0; strray[i] != '\0'; i++)
    {
        bool *isOccupied = &flag[i];
        if (!*isOccupied)
        {
            result[k] = strray[i];
            *isOccupied = true;
            permute(strray, k + 1);
            *isOccupied = false;
        }
    }
}
```

```
int main()
{
    char s[] = "ABC";
    permute(s);
    return 0;
}
```

- Permutations of a String with swapping
  - 0th index is variable `l` and (length-1)th index is variable `h` and `i` is a local variable equal to `l` that increments till it is equal to (length-1)
  - swap element at `l` and element at `i`(which might be the same)
  - recursive call with `l+1` instead
    - base condition is to print string permutation when `l==i`



Recursion Tree for Permutations of String "ABC"

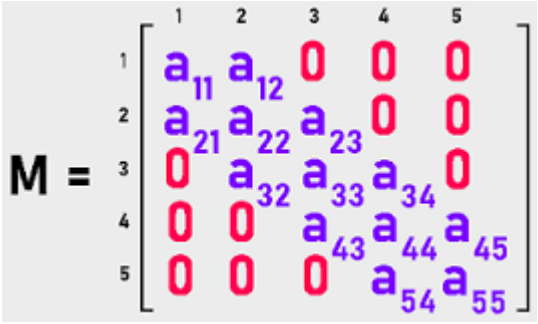
```
#include <iostream>
#include <cstring>
using namespace std;

void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}

void permute(char strray[], int h, int l = 0)
{
    if (l == h)
    {
        cout << strray << endl;
        return;
```

```
    }
    for (int i = l; i <= h; i++)
    {
        swap(strray[l], strray[i]);
        permute(strray, h, l + 1);
        swap(strray[l], strray[i]);
    }
}

int main()
{
    char s[] = "ABC";
    permute(s, strlen(s) - 1); // without -1 we will end up swapping
'\0' as well so the elements after it will be ignored by cout
    return 0;
}
```

Matrices

- assuming square matrices are of order n start from index 1 and arrays start from index 0 so i, j is actually i-1, j-1
- 2-D arrays can quickly get inefficient and end up wasting memory by storing information we don't need. Still they can be quickly read due to contagious memory allocations
- a diagonal matrix has all non diagonal elements equal to zero and the amount of space wasted increases as we increase order of a diagonal matrix
  - which can instead be represented by a linear array of just the diagonal elements
  - so when i equals j then use ith or jth index of the linear array else use 0
  - as performing operations on a diagonal matrix introduces lot of redundancy in the operations as we perform more operations on the zeroes instead of the diagonal elements
- a lower triangular matrix has all elements above the diagonal equal to zero(excluding the diagonal as well)
  - element at i,j is 0 when `i<j` and non-zero for `i ⩾ j`
  - total number of non zero elements is `n(n+1)/2` and total number of zero elements are `(n^2)-(n(n+1)/2) = n(n-1)/2`
  - to store the elements below diagonal and including diagonal elements we need a linear array of size `n(n+1)2`

- 1st row of matrix has one non zero element, 2nd has 2, 3rd has 3 and so on ◺
    - the 1st element of the linear array is the 1st element of the 1st row
    - the 2nd element of the linear array marks the start of the 2nd row and the 3rd element marks the end
    - the 4th element of the linear array marks the start of the 3rd row and the 5th element marks its end
- i,j(they start from 1 not 0) of the 2-D matrix can be mapped to `(i(i-1)/2)+j-1`th index of the linear array
    - the above mapping is called row major mapping as we fill the linear array by scanning the matrix from left to right then top to bottom like breadth first or row by row
- in column major mapping we fill the array column by column(top to bottom) then left to right
    - 1st column has n elements 2nd has n-1 and 3rd has n-2 and so on ◺
        - the 1st element of the linear array is the first element of the 1st column and the nth element of linear array marks the end of the 1st column
        - the (n+1)th element of the linear array marks the start of the 2nd column and is the 1st element of the 2nd column and the 2nd column ends at (n+n-1)th element of the linear array
    - i,j(they start from 1 not 0) of the 2-D matrix can be mapped to `(n(j-1)-((j-2)(j-1))/2)+i-j`th index of the linear array
        - the above mapping is called column major mapping
- an upper triangular matrix has all elements below the diagonal equal to zero(excluding the diagonal as well)
    - element at i,j is 0 when `i>j` and non-zero for `i⩽j` (sign flipped)
    - total number of non zero elements is `n(n+1)/2` and total number of zero elements are `(n^2)-(n(n+1)/2) = n(n-1)/2` (same as lower triangular matrix)
    - to store the elements below diagonal and including diagonal elements we need a linear array of size `n(n+1)2` (same as lower triangular matrix)
    - row major mapping: `(n(i-1)-((i-2)(i-1))/2)+j-i` (interchange i and j of column major mapping of lower triangular matrix)
    - column major mapping: `(j(j-1)/2)+i-1` (interchange i and j of row major mapping of lower triangular matrix)

- symmetric matrices: when element at i,j is equal to the element at j,i then the matrix is symmetric
  - so we don't need to store duplicates, we can discard either the lower triangle or the upper triangle(always including the diagonal as for every diagonal element `i=j` so they are unique)
- tridiagonal matrix
  - 

    - as we can see for the diagonal elements `i=j` so i-j=0
      - similarly for the lower diagonal matrix i-j=1
      - and for upper diagonal matrix i-j=-1
      - we can say that for a non zero element `|i-j| ⩽ 1` and for a zero element `|i-j| > 1`
    - total number of non zero elements are `3n-2`
    - the linear array is first populated with lower diagonal elements followed by center diagonal elements followed by upper diagonal elements
    - mapping non zero elements to a linear array
      - when i-j=1 then i,j of matrix is mapped with index i-1 of the linear array
      - when i-j=0 the i,j of matrix is mapped with index n-1+i-1 of the linear array
      - when i-j=-1 the i,j of matrix is mapped with index 2n-1+i-1 of the linear array