# 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)`
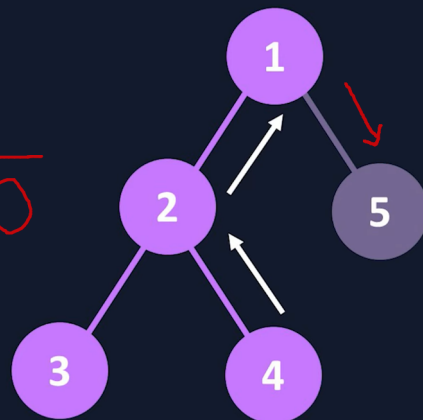
  - Breadth First Search for trees and graphs
    - you don't go to deepest point like DFS
    - instead you make sure that the sibling node has been visited
    - once you are on a node look at its children and add them to a queue and then you visit the node in the queue and add them to visited array 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
  - 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 less 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 if 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 restrictive

- 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);
    }
```
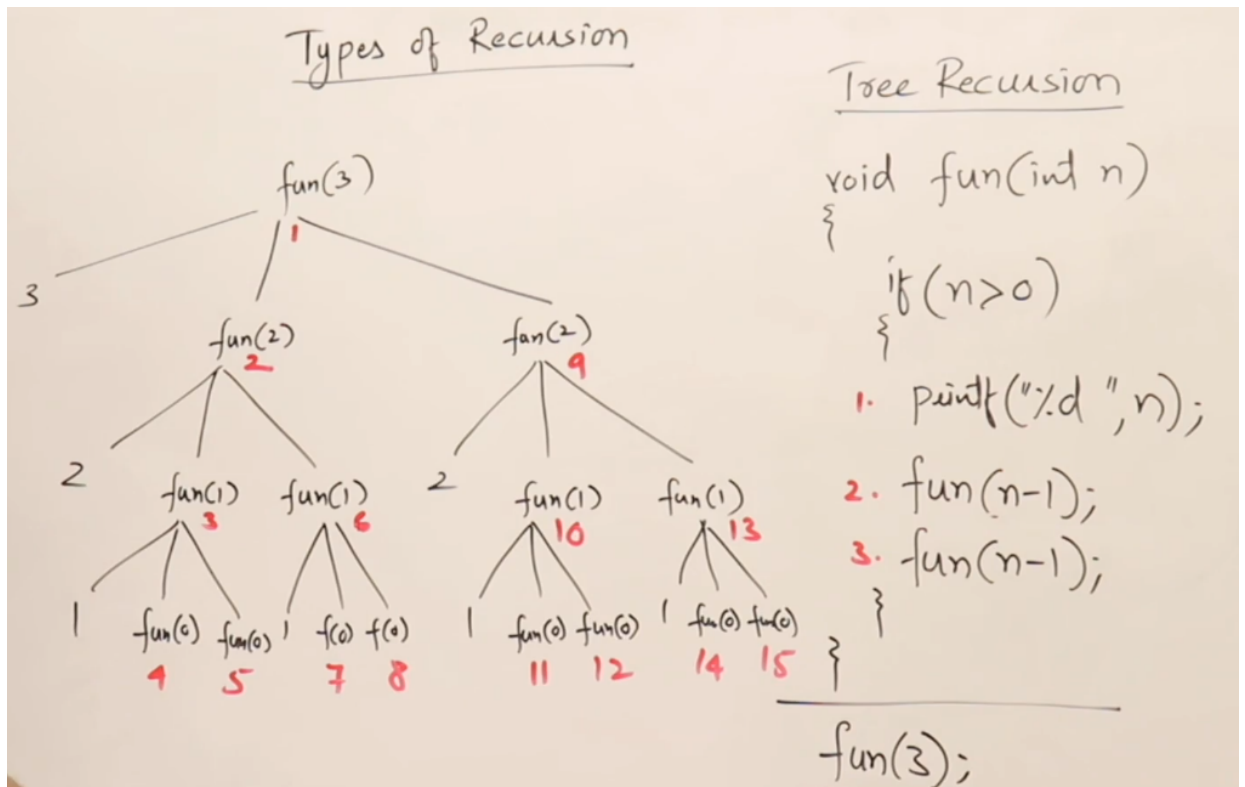
```
}

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


Types of Recursion

Tree Recursion

```
void fun(int n)
{
    if (n>0)
    {
        1.  printf("%d ",n);
        2.  fun(n-1);
        3.  fun(n-1);
    }
}

fun(3);
```

-
- 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);
    }
}
```
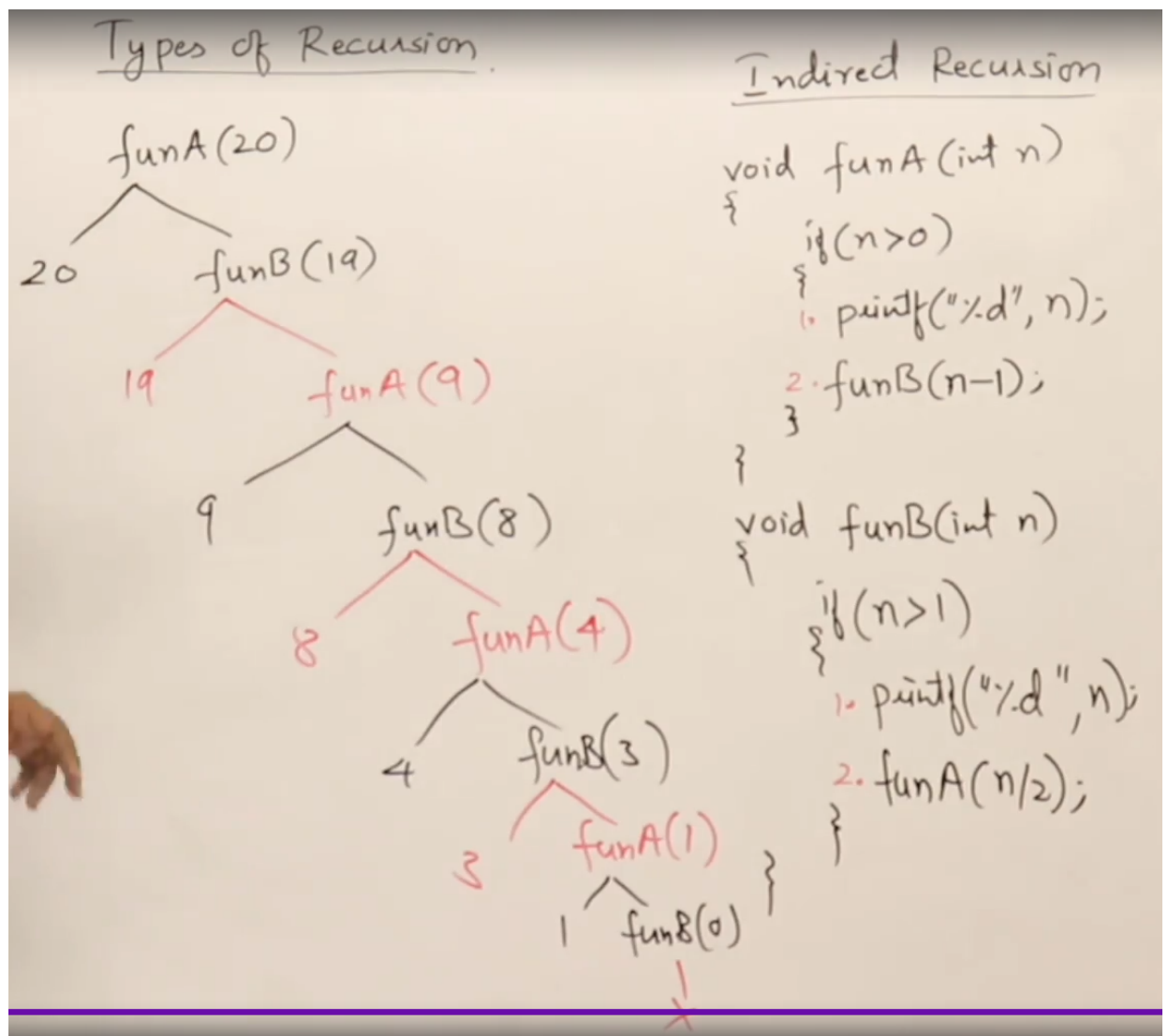
```cpp
void funB(int n)
{
    if (n > 1)
    {
        cout << "\n"
            << n << " ";
        funA(n / 2);
    }
}

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

Types of Recursion

funA (20)

20   funB (19)

    19   funA (9)

        9   funB (8)

            8   funA (4)

                4   funB (3)

                    3   funA (1)

                        1   funB (0)

Indirect Recursion

```
void funA (int n)
{
    if (n > 0)
    {
1.      printf("%d", n);
2.      funB(n-1);
3.
    }
}

void funB(int n)
{
    if (n > 1)
    {
1.      printf("%d ", n);
2.      funA(n/2);
    }
}
```

- nested

- parameter of a recursive call function is the same function

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#include <iostream>
using namespace std;
float e(int x, int n)
{
    int res = 1;
    for (; n > 0; n--)
```

```cpp
        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

```cpp
#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)`
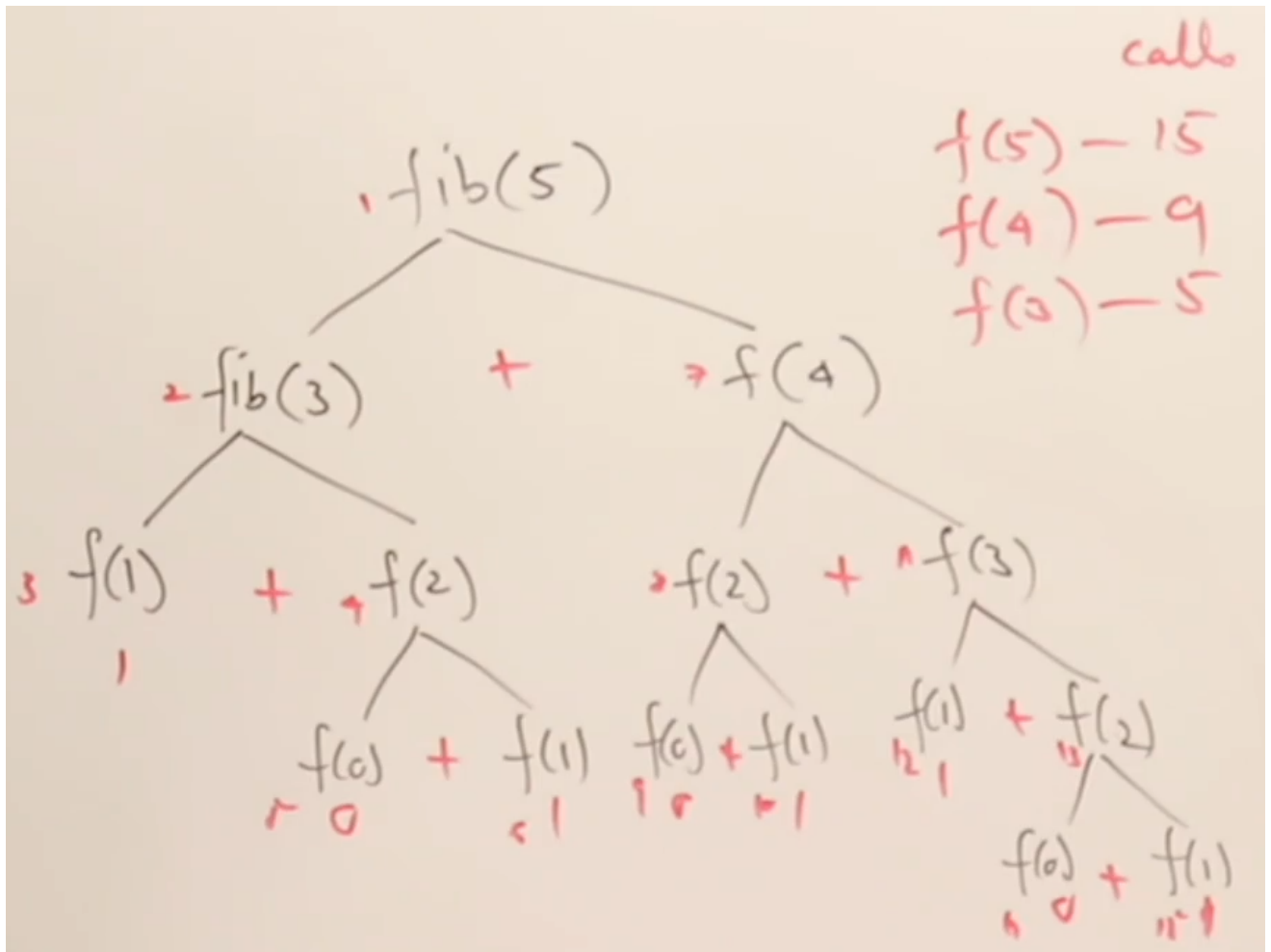
```cpp
#include <iostream>
using namespace std;
int fibo(int n)
{
```

```
    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)`

```cpp
#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

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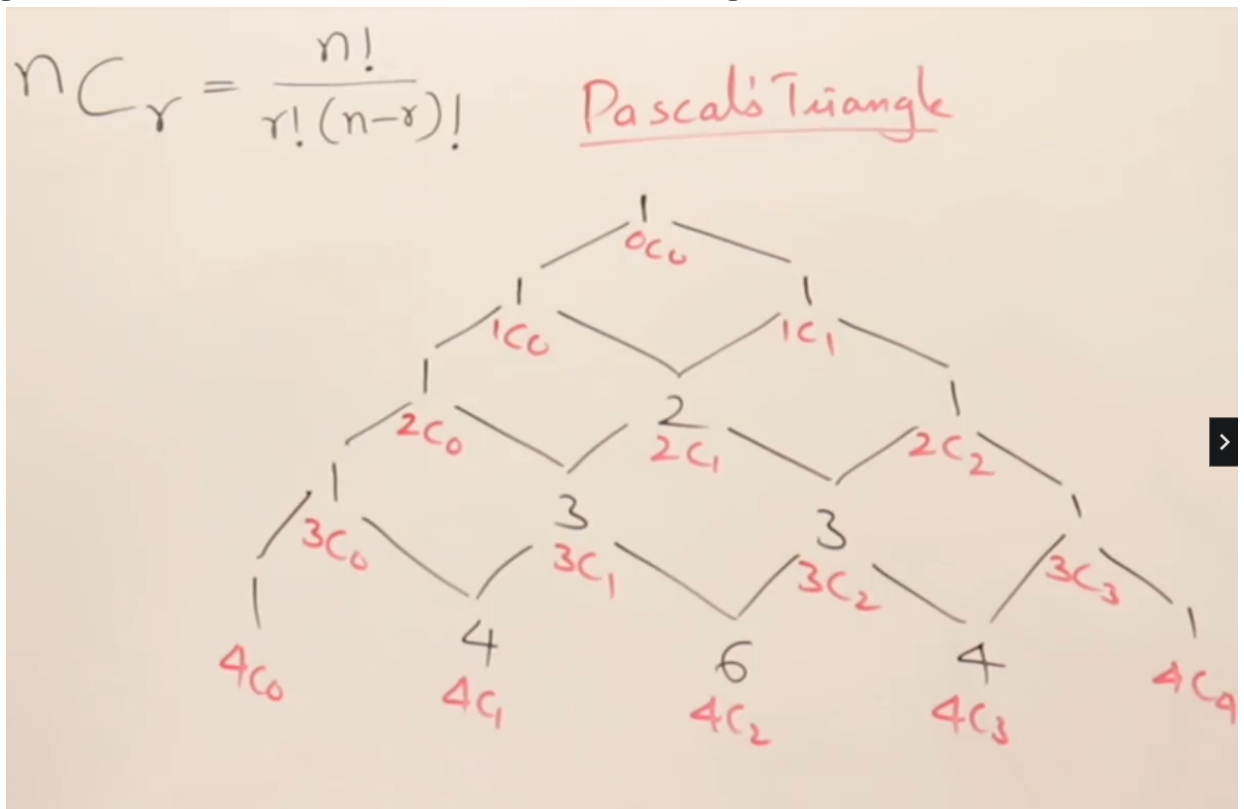
```cpp
    int n, r;
    cin >> n >> r;
    if (r > n)
    {
        cout << "Invalid Input " << r << " grtr than " << n;
        return -1;
    }
    cout << n << "C" << r << " = " << fac(n) / (fac(r) * fac(n -
r));
    return 0;
}
```

- using recursion we need to use Pascal's Triangle

$$nC_r = \frac{n!}{r!\,(n-r)!}$$

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
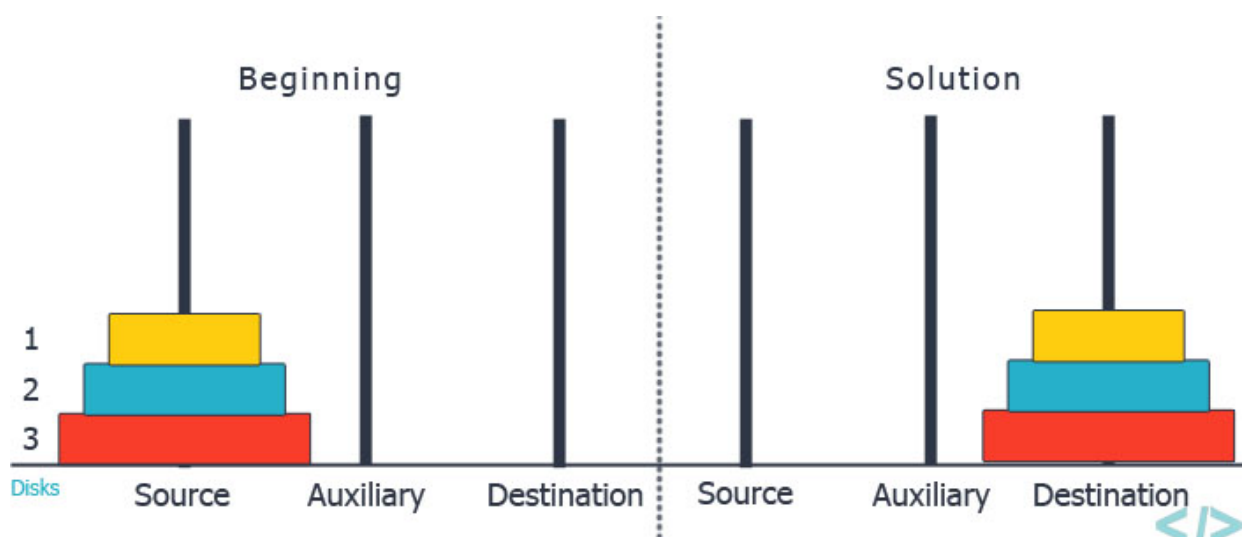   top to bottom when returning

```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;
}
```
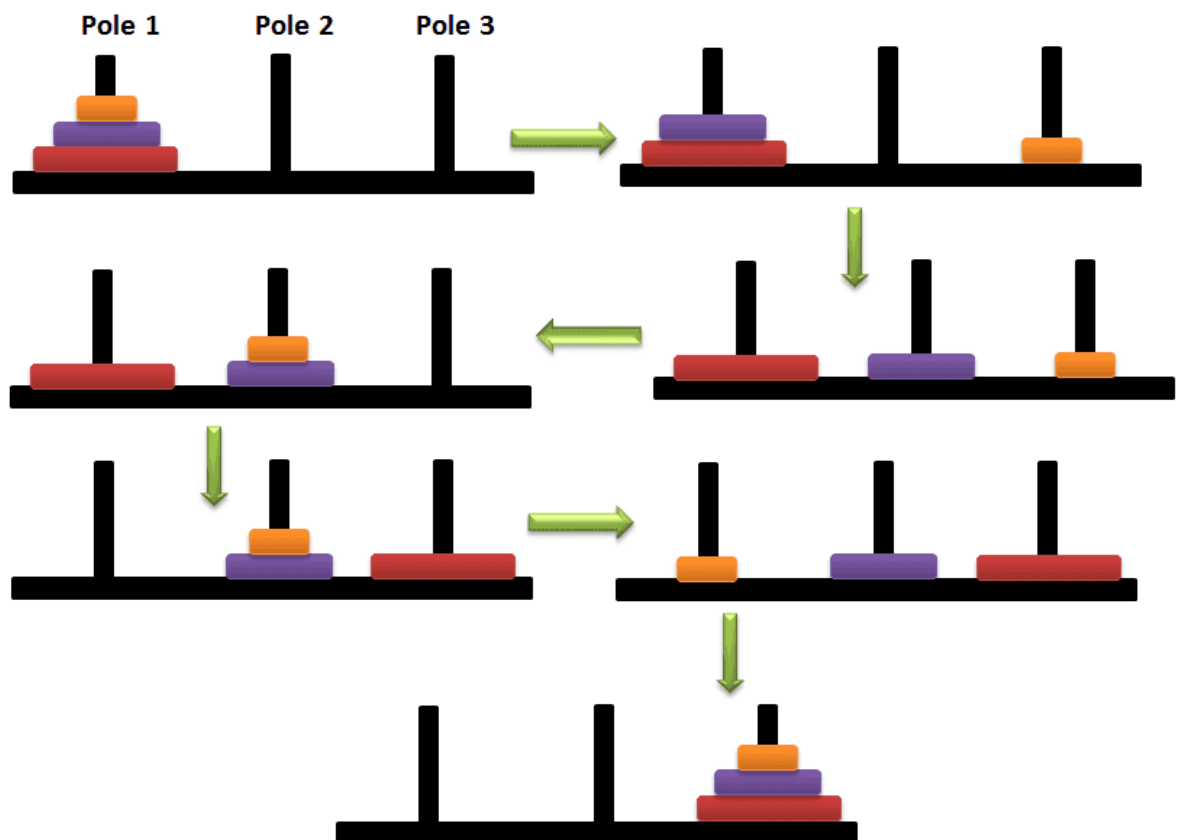
- Tower of Hanoi

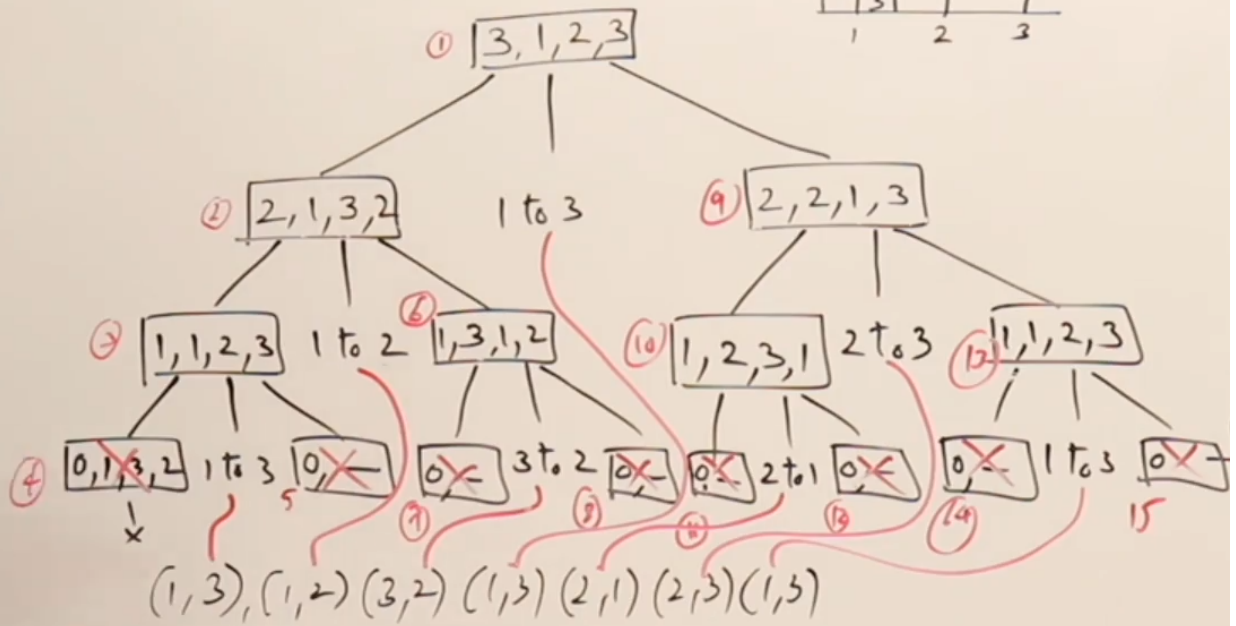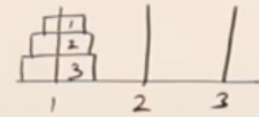   - Question:

    - 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

Tower of Hanoi

① 3,1,2,3

1 to 3

② 2,1,3,2

⑨ 2,2,1,3

③ 1,1,2,3   1 to 2   ⑥ 1,3,1,2

⑩ 1,2,3,1   2 to 3   ⑫ 1,1,2,3

④ 0,1,3,2   1 to 3   0   0   3 to 2   0   0   2 to 1   0   0   1 to 3   0

(1,3),(1,2) (3,2) (1,3) (2,1) (2,3)(1,3)

```cpp
#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;
    toh(n - 1, sp1, dp3, mp2);                              //
move (n-1)th disc from sp1 to mp2 using dp3 as Auxiliary Pole
    cout << "moved 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;
}
```