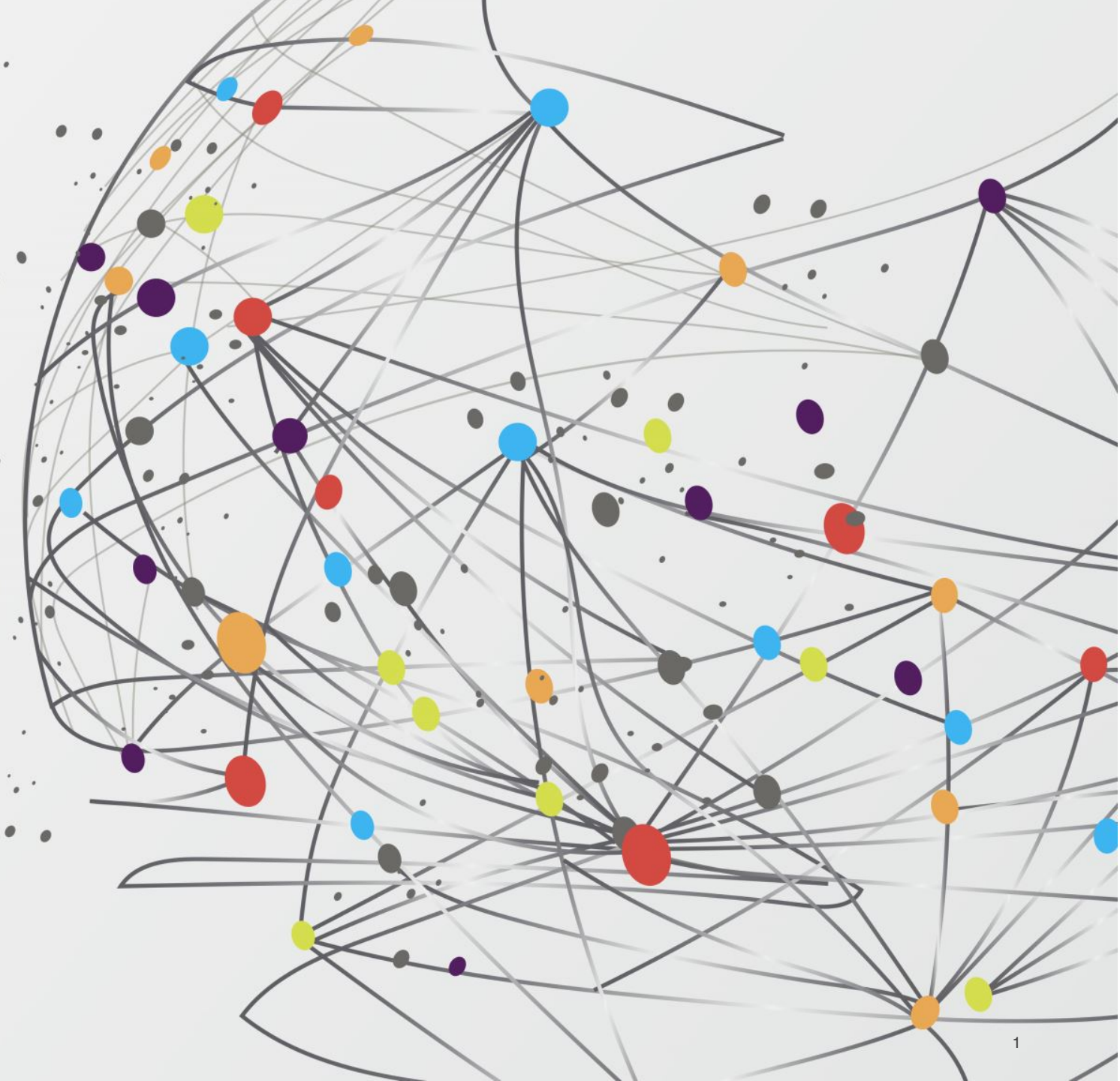
WEEK 2

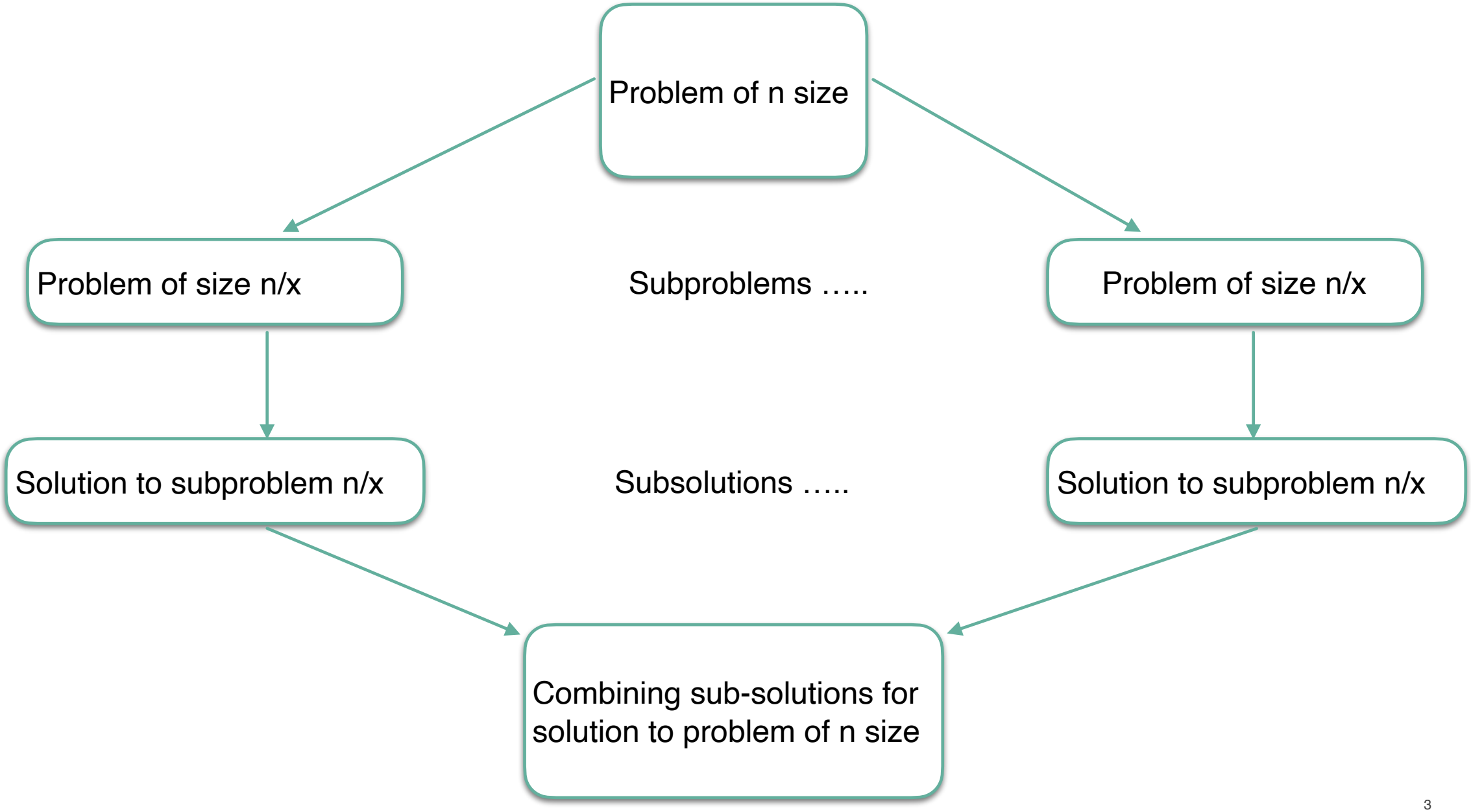


NEWTON SCHOOL

RECURSION : DIVIDE AND CONQUER

- Divide : Breaking the problem into sub problems that are smaller instances of same type of problem, util they become simple enough to be solved directly.
- Recursion : Recursively solving these sub problems which are of same type.
- Conquer : Appropriately combining these answers to give solution to original problem.

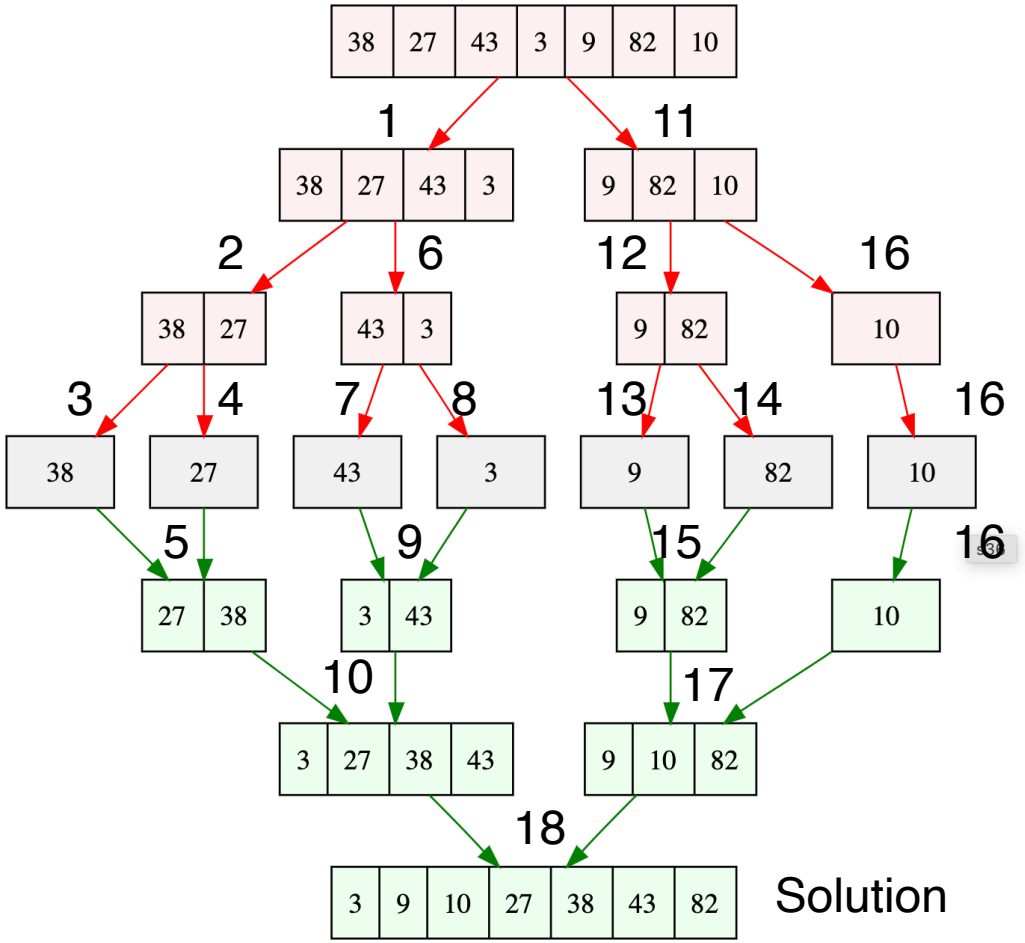
- Adv: Solve difficult problems, parallelism
- DisAdv : Stack memory required for storing recursive calls, Not better always than iterative approach for eg summation of numbers.
- Eg: Binary Search, Merge Sort


Problem of n size
Problem of size n/x
Subproblems …..
Problem of size n/x
Solution to subproblem n/x
Subsolutions …..
Solution to subproblem n/x
Combining sub-solutions for solution to problem of n size

MERGE SORT - UNDERSTANDING THE ALGO

- It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.
- Two parts of Algo:
- 1) Recursively dividing the array till size becomes 1.
- 2) Merging two halves.

MERGE SORT - UNDERSTANDING THE ALGO



MERGE SORT - COMPLEXITY

- Worst case/Best case/Average case time complexity -> O(nlogn) as merge sort always divides the array into two halves and takes linear time to merge two halves.
- Every step is having -> n operations, Total number of steps are -> logn , Hence time complexity is O(nlogn).
- Worst case space complexity -> O(n) auxiliary space

RECURSION : MASTER THEOREM

- In D&C , we solve all subproblems recursively.
- These recursive problems can easily be solved using master theorem.
- This theorem is used to determine running time of algorithms (divide and conquer algorithms) .
- This theorem gives the time complexity of recurrence relation , if our recurrence relation is in this form:
- T(n) = aT(n/b) + f(n) : Time required for n size problem
- where n = size of the problem

a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
f(n) = cost of work done outside the recursive calls like dividing into subproblems and cost of combining them to get the solution.

RECURSION : MASTER THEOREM

For the Recurrence: $T(n) = aT(n/b) + \Theta(n^c)$, $a >= 1$, $b > 1$

There are following three cases:

1. If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$

2. If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$

3. If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

RECURSION : MASTER THEOREM

1. $T(n) = 2\,T(n/2) + \Theta(n)$

$a = 2, b = 2, c = 1$

$\rightarrow c = \log_b a$

Time Complexity: $\Theta(n \log_2 n)$

- Merge Sort
- We will discuss theorem's advance version in time complexity

1-D ARRAY

- An array is a collection of similar type of elements which has contiguous memory location.
- **Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
- Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.
- We can store primitive values or objects in an array in Java.
- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

1-D ARRAY

- There are two types of array.
- **Single Dimensional Array** , Multidimensional Array
- dataType[] arr; (or)
- dataType []arr; (or)
- dataType arr[];
- Eg: arrayRefVar=**new** datatype[size];

MATHS

- Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

PRIMES

- A prime number is a natural number greater than **1**, which is only divisible by 1 and itself. First few prime numbers are : 2 3 5 7 11 13 17 19 23 …..
- Two is the only even Prime number.
- Every prime number can be represented in form of 6n+1 or 6n-1 except the prime number 2 and 3, where n is a natural number.
- Two and Three are only two consecutive natural numbers that are prime.

PRIMES

- Check whether a number is Prime or not?
- Find prime numbers between two numbers ?

SET OF DIVISORS

- Given a natural number n, print all distinct divisors of it.
- **Input:** n = 100
- **Output:** 1 2 4 5 10 20 25 50 100

MODULO ARITHMETIC

- Modular arithmetic is the branch of arithmetic mathematics related with the “mod” functionality.
- Modular arithmetic is related with computation of “mod” of expressions.
- Expressions may have digits and computational symbols of addition, subtraction, multiplication, division or any other.
- a = b x q + r where 0 <= r < b, for any pair of integers a and b (b is positive)

MODULO ARITHMETIC

- (a + b) mod m = ((a mod m) + (b mod m)) mod m
- (a x b) mod m = ((a mod m) x (b mod m)) mod m
- (a / b) mod m = (a x (inverse of b if exists)) mod m :The modular inverse of a mod m exists only if a and m are relatively prime i.e. gcd(a, m) = 1.
- Hence, for finding inverse of a under modulo m, if (a x b) mod m = 1 then b is modular inverse of a.

a = 5, m = 7
(5 x 3) % 7 = 1
hence, 3 is modulo inverse of 5 under 7.

MODULO EXPONENTIATION (POWER IN MODULAR ARITHMETIC)

- Finding a^b mod m is the modular exponentiation. There are two approaches for this – recursive and iterative.
- a = 5, b = 2, m = 7

```
(5 ^ 2) % 7 = 25 % 7 = 4
```

MODULO EXPONENTIATION (POWER IN MODULAR ARITHMETIC)

```
power(x,y)
{
    res=1
    while(y>0)
    {
        If y is odd
            res=res*x
        y=y/2
        x=x*x
    }
    return res
```

```
power(2,8)
    |
res=1,y=4,x=4
    |
res=1y=2,x=16
    |
res=1,y=1,x=256
    |
res=256,y=0,x=65536
```

MODULO EXPONENTIATION (POWER IN MODULAR ARITHMETIC)

- The problem with above solutions is, overflow may occur for large value of y or x. Therefore, power is generally evaluated under modulo of a large number.
- That’s why we calculate (x(pow y))mod p

MODULO EXPONENTIATION (POWER IN MODULAR ARITHMETIC)

- **static int** power(**int** x, **int** y, **int** p)

```
{
  int res = 1;

  x = x % p;
  if (x == 0)
    return 0;

  while (y > 0)
  {
    if ((y & 1) != 0)
      res = (res * x) % p;

    y = y >> 1; // y = y/2
    x = (x * x) % p;
  }
  return res;
}
```

QUESTIONS

THANKS