Recursion

What is Recursion?

→ Function calling itself.

→ solving a problem using smaller problems ⇒ called Recursion.

sum(n) ⇒ Returns sum of n natural numbers

sum(n) = 1 + 2 + 3 + ... n-2 + n-1 + n

sum(n-1)

sum(1) = 1

sum(n) = n + sum(n-1)

if(n == 1)

sum(4)
↳ 4 + sum(3)
↳ 3 + sum(2)
↳ 2 + sum(1) return 1;
↳ 1 + sum(0)
↓
0 + sum(-1)
↓
-1 + sum(-2)
↓
Infinite Loop
+ stack overflow

Base / Terminating

```
int sum(N) {
    if(N == 1) return 1;
    return N + sum(N-1);
}
```

3 Steps of Recursion

your function

1) Assumption : What do you do.

sum(n) ⇒ (Return the sum of n natural numbers)

2) Main Logic / Recursive Relation :

Solve the sub-problems bigger problem using smaller

sum(n) = n + sum(n-1);

3) Base / Termination condition

When do we want our function to stop

if(n == 1) return 1;

- Example : Find N!

6! = 6 * 5 * 4 * 3 * 2 * 1

Re-writing the problem

Fact(N)

Assumption : Fact(N) returns N!

Main Logic : Fact(N) = N * Fact(N-1)

Termination condition:

if(N == 1) return 1;

Fact(6) = 6 * 5 * 4 * 3 * 2 * 1

Fact(5)

fact(6) = 6 * fact(6-1)

0! = 1

```
int fact(N)
{
  if(N<0)
    return ERROR;
  if(N==1 || N==0)
    return 1;
  return N * fact(N-1);
}
```

fact(0)

n = -1

0! = ? 1

fact(5)

1! = 1
0! = 1

fact(0)

Question: Print numbers from i to n in increasing order.

i ≤ N

F(3, 7) = 3 4 5 6 7

F(1, 6) = [1 2 3 4 5 6]

(1) F(2,6)

Assumption:

print_inc(i, n) = Print numbers from i to n in increasing order.

Main Logic:

```
print(i);
print_inc(i+1);
```

(i) i+1 i+2 ..... n

print_inc(i+1, n)

Termination condition:

```
if( i > n)
    return;
```

F(1,3): 1 2 3

(1) → F(2,3)

(2) → F(3,3)

(3) → F(4,3)

i > n

```
void printinc ( i, n) {
    if( i > n) return;
    print(i);
    print_inc (i+1, n);
}
```

print_in (i+1, n)

Question: Print numbers (i, n) in decreasing order

F(3, 6) : 6 5 4 3

```
print_dec (i+1, n)
print(i);
```

3 (4 5 6)

1 1 6 5 4

F(3, 6)

F(3, 6)

F(3, 7) = 7 6 5 4 3

i n

printdec (i, n) = ?

Assume.

Print numbers in decreasing order.

i = 3, N = 7

3 4 5 6 7

print_dec (4, 7) = 7 6 5 4 3

Question: Find a^b

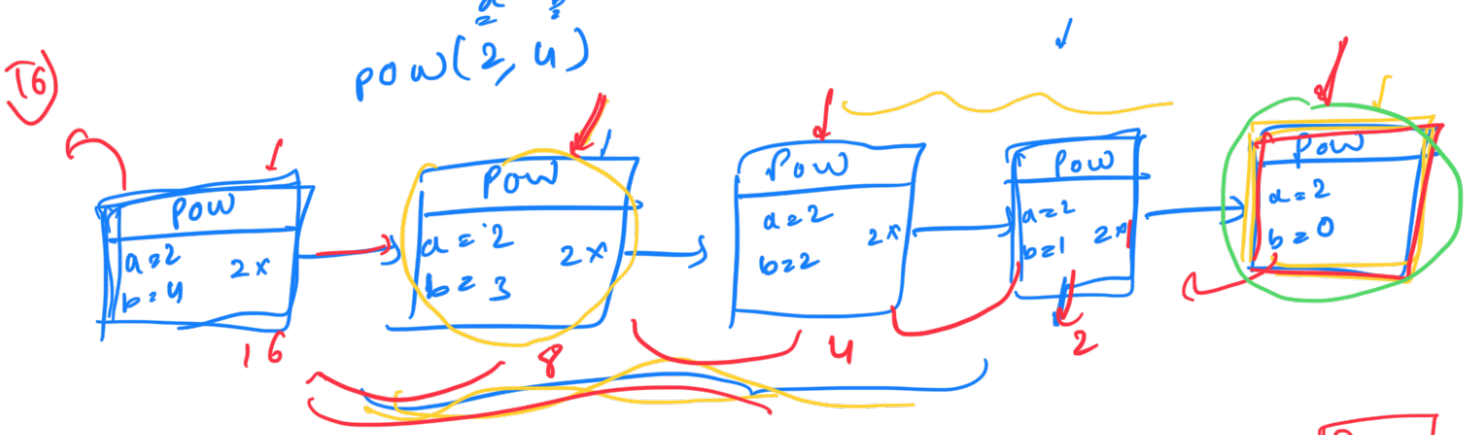
$$a^b = a \times a^{b-1}$$

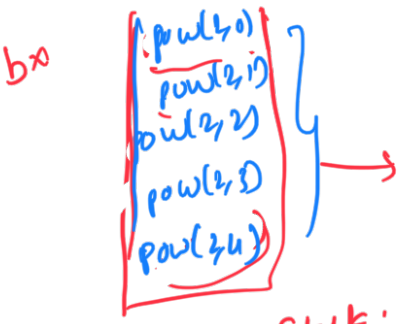
Assumption: pow(a, b) should a^b

Main Logic: pow(a, b) = a x pow(a, b-1);

Base condition: if (b == 0) return 1;

How these functions are being traced:





Recursion stack.

LIFO = Last In First Out

(b+1) Functions

S.C: O(b)

(b+1) x 1 = O(b);

Pyramid

Space Complexity:

=> O(Maximum no of active functions in stack * S.C per function)

O(1) { int pow(a, b) { if(b == 0) return 1;

T.C: O(1)

S.C: O(b)

```
{
    return a * pow(a, b-1);
}
```

S.C per Function: O(b)

```
{
    int arr[b];
    // Processing
}
```

T.C: O(b * b) = O(b^2)

YES, all recursive problems can be solved using iterations

→ Small and Elegant

Fast Exponentiation

Time complexity:

O(Total no. of functions called * T.C per function)

T.C: O(b)

Fib(1) = Fib(6) + Fib(5)

Fib(N) = Fib(N-1) + Fib(N-2)

Question: Fibonacci series

1 1 2 3 5 8 13 21 34 55 89

Fib = 1 1

N=1 N=2

Fib(n) = Fib(n-1) + Fib(n-2)

Fib(1) = 1

Fib(2) = 1

Assumption: Fib(N) return the N^th number in Fibonacci series

Main Logic:

Fib(n) = Fib(n-1) + Fib(n-2)

2 Base Case

Base condition:

if(N == 1 || N == 2)
return 1;

Usually need n(base) = n (function calls)

```
if (N<=2)
    return 1;
```

1 Base Case

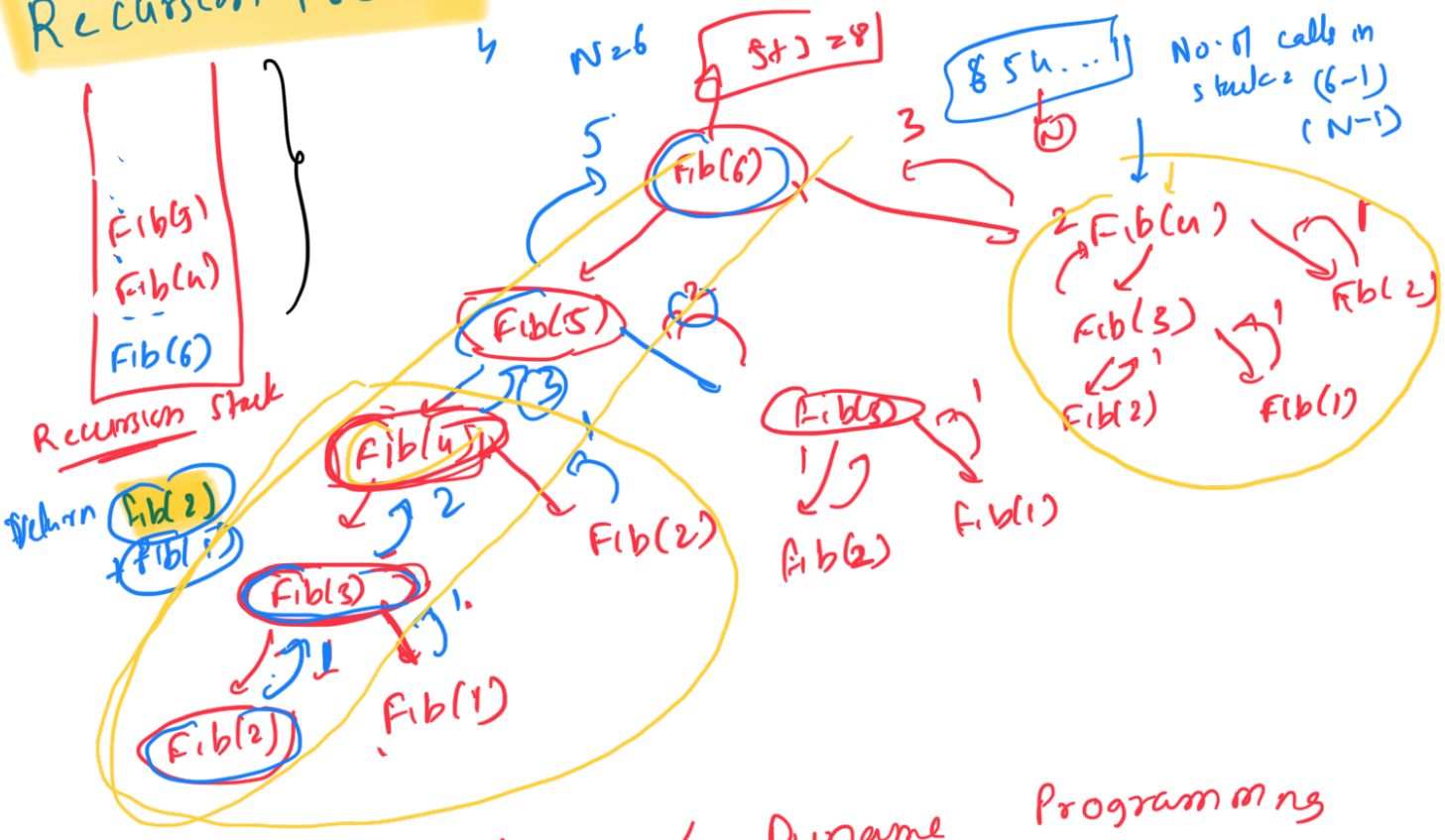
Program Execution is sequential.

n ≥ 1

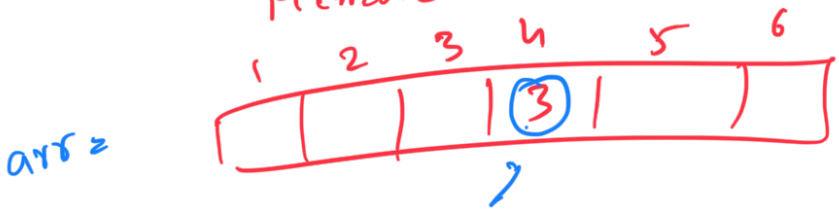
T.C. O(1)

```
int Fib(n) {
    if n == 1 || N == 2 ) return 1;
    return Fib(n-1) + Fib(n-2);
```

Recursion Tree



Memoization / Dyname Programming



S.C: O(Height of Recursion Tree × S.C per Function)

h = n-1

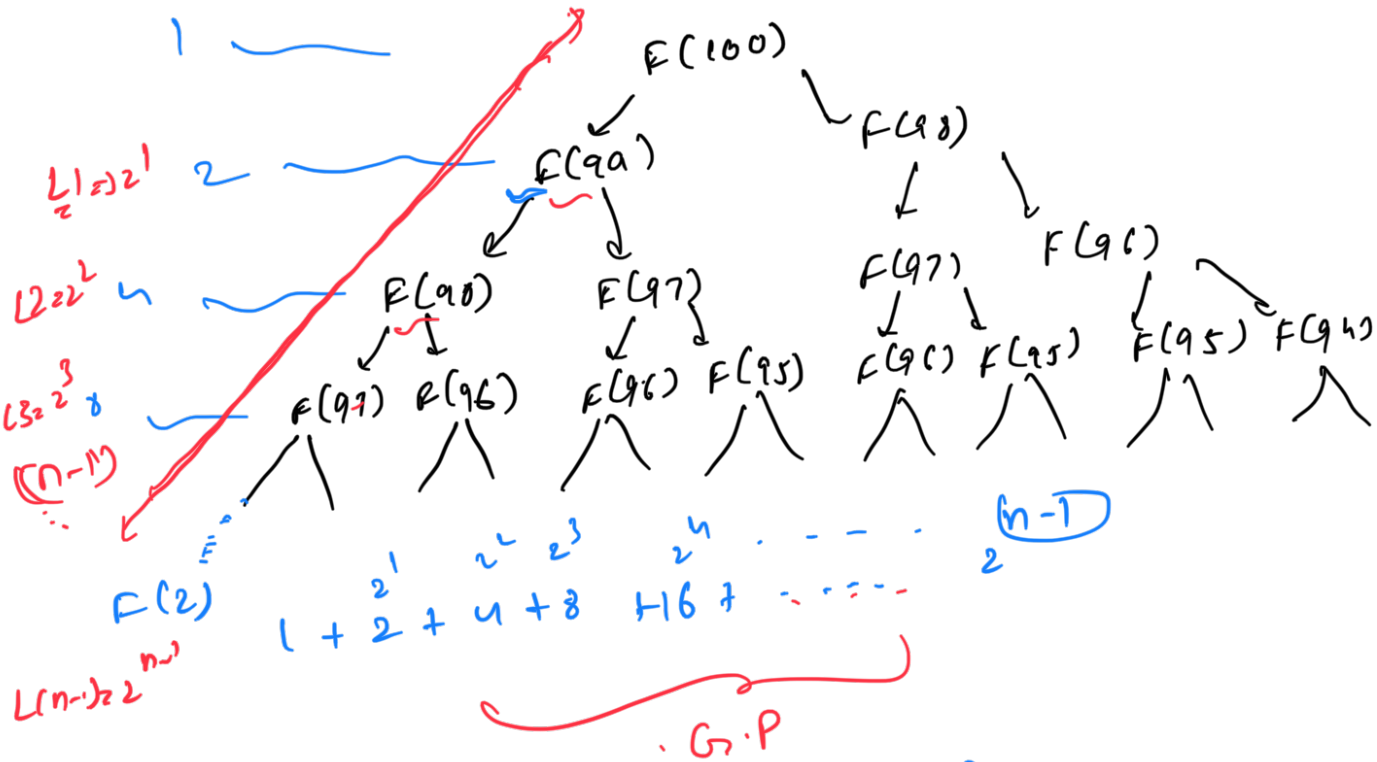
S.C: O(n × 1)

S.C: O(n)

Time Complexity:

O(# function calls × T.C per Function)

O(1)



a, ar, ar^2, ar^3 . . . .

No. of Terms

(n-1) Terms

$\frac{a(r^n - 1)}{r - 1}$

$\frac{1(2^{n-1} - 1)}{2 - 1} = 2^{n-1} - 1 = \frac{2^n}{2} - 1$

Const

nodes = $O(2^n)$

T.C: $O(2^{n+1})$ = $O(2^n)$

Tail Recursion:

If the Recursive call is the last thing to be executed by function, then it is called Tail Recursion.

```
void solve(n){
    if(n==0) return n;
    print(n);
    solve(n-1);
}
```

solve(2) → solve(1)

Output = 2

Tail Recursion

solve(2) → solve(1) → solve(0)

2 1

sole(1)

Recursive call

F1 → F2 → F3 (Sequential)

```
int solve(n){
    if(n==0) return 0;
    print(n);
    return 1 + solve(n-1);
}
```

✗ Tail Recursion

(+1) will be executed once solve(n-1) is computed.

Why Tail Recursion:

Space Complexity: O(1)

→ We won't use Recursion stack.

→ Compilers to recognise this and they are not stored in stack.

fib(n-1)+fib(n-2)

$T(n) = O(1) + O(1) + T(\frac{2n}{3})$ S.C: $O(1)$

$T(n) = T(\frac{2n}{3}) + 2$ ⇒ $T(n) = T(\frac{2n}{3}) + O(1)$

Time complexity

Multiple Recursion calls

✗ Tail Recursion

```
void solve(n) {
    if(n <= 1) return;
    print(n);
    solve(2*n/3);
}
```

Time taken for a function

$T(n) = T(\frac{2n}{3}) + O(1)$

$T(n) = T(\frac{2n}{3}) + O(1)$

$O(1)$

$T(n) = T(\frac{2n}{3}) + O(n)$

```
for(i=0; i<n; i++)
    sum = sum + i;
```

$T(n) = O(1) + O(n) + O(1) + T(\frac{2n}{3})$

$T(n) = T(\frac{2n}{3}) + O(n)$

$n = 3$

$F(3) \rightarrow F(2) \rightarrow F(1)$

3 Functions

$F(2*\frac{3n}{3})$

$F(2*\frac{2}{3})$

$O(n)$?

$n = 10$

$F(10) \rightarrow F(6) \rightarrow F(4) \rightarrow F(2) \rightarrow F(1)$

4 Functions

$n \rightarrow \frac{2n}{3} \rightarrow (\frac{2}{3})^2 n \rightarrow (\frac{2}{3})^3 n \dots (\frac{2}{3})^k n$

$(\frac{2}{3})^k n \rightarrow 1$

K steps

$(\frac{2}{3})^k n = 1$

$n = (\frac{3}{2})^k$

$\log_2^n = k \log_2^{3/2}$

$\frac{\log_b^a}{\log_c^a} = \log_c^b$

Property of log

$k = \log_{3/2}^n$

$O(\log n)$

$k = \frac{\log 2}{\log \frac{3}{2}} \Rightarrow \log_{3/2} 2$

$k = \log_{3/2} n$.

T.C: $O(\log_{3/2} n) = O(\log n)$

Master's theorem

→ Technique to find T.C of certain Recursive functions.

if $T(n) = a\,T\left(\frac{n}{b}\right) + O(n^d)$

$a > 0$, $b > 1$, $d \geq 0$

case 1: $b^d > a \rightarrow O(n^d)$

case 2: $b^d = a \rightarrow O(n^d \log_b n)$

case 3: $b^d < a \rightarrow O(n^{\log_b a})$

$T(n) = a\,T\left(\frac{n}{b}\right) + O(n^d)$

$T(n) = T\left(\frac{2n}{3}\right) + O(1)$

$T(n) = T(n/(3/2)) + O(n^0)$

$a = 1$

$b = 3/2$

$d = 0$

$b^d = \left(\frac{3}{2}\right)^0 = 1$

$a = 1$

$b^d = a$

$O(n^d \cdot \log_b n)$

$O(\log_{3/2} n)$

$O(n^0 \cdot \log_{3/2}) = O(\log_{3/2})$

Question:

```
int solve(n) {
    if (n <= 1) return n;
    return solve(n/2) + solve(n/2);
}
```

$T(n) = O(1) + T(\frac{n}{2}) + T(\frac{n}{2})$

$T(n) = 2T(\frac{n}{2}) + O(1)$

$a = 2, \; b = 2, \; d = 0$

$b^d = 2^0 = 1$

$a > b^d$

$O(n^{\log_b^a}) = O(n^{\log_2^2})$

$= O(n)$

Back tracking

Generating all subsets / subsequences / combinations or permutations using recursion is called backtracking.

Backtracking is BRUTE FORCE

↓

Implemented using Recursion

Question: Given an array, check if there

Question:

is a subsequence of sum = k

A = [10, 1, -2, 7, 6, 1, 5]

k = 8

True / False

- [1, 7]
- [2, 6]
- [1, 2, 5]
- [2, 1, 5]
- [1, 6, 1] → {1, 1, 6} } Subseq
  {6, 1, 1}

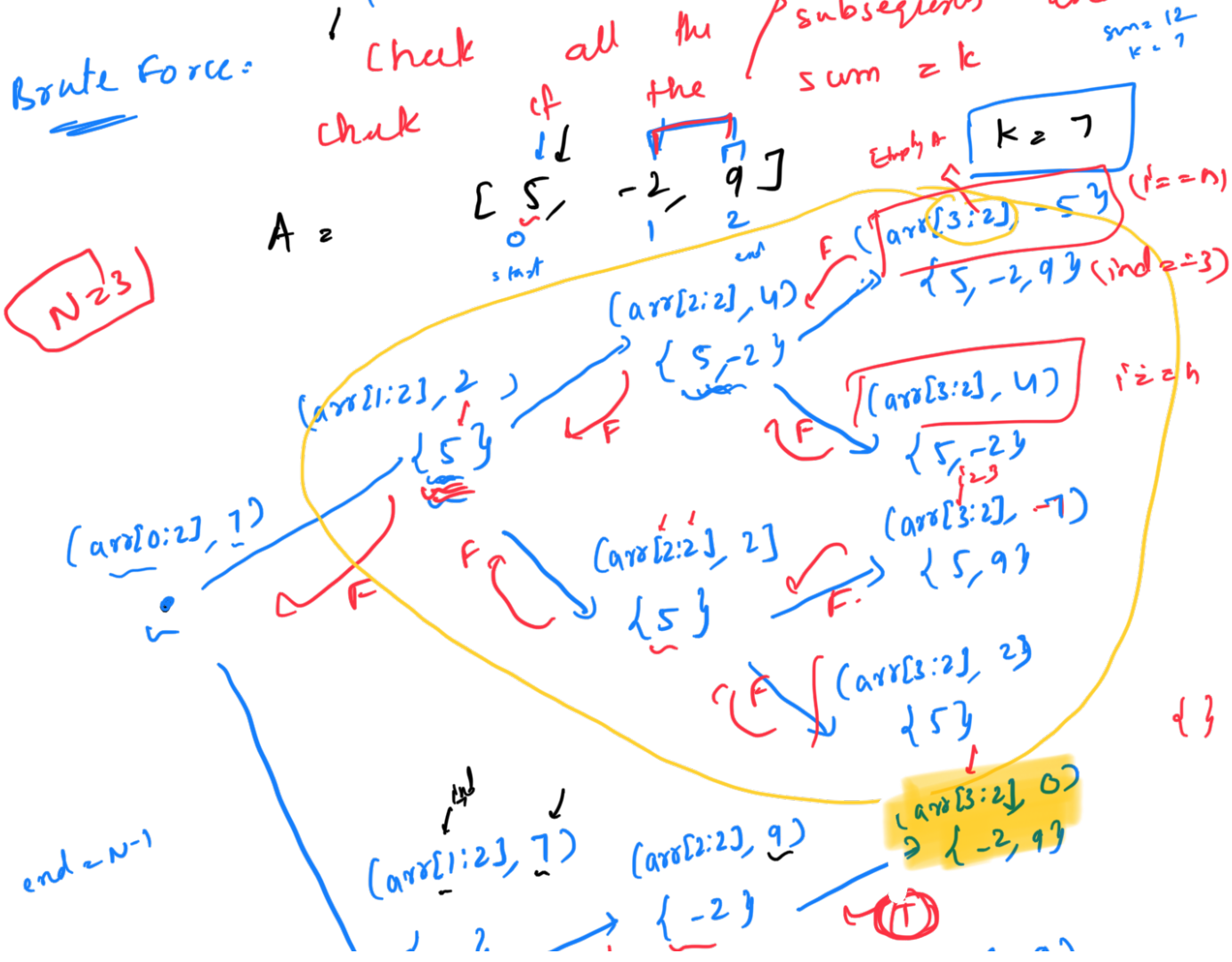
A = [5, -2, 9]

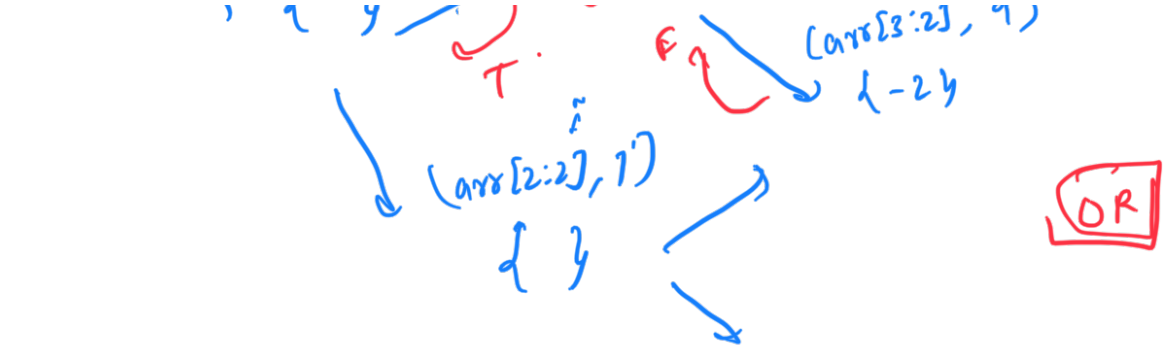
k = 7

{-2, 9} = 7

{2 x 2 x 2} = 2^3

Brute Force: Check all the subsequences and check if the sum = k





Parameters:

ind ⇒ Index of element that we are considering

n ⇒ Size of array

k ⇒ sum required

arr ⇒ ith element

1) Include → arr[i]

F(i , k) ⇒ F (i+1, k-arr[i]);

⇒ We want a sum of k

{ arr[i]

k - arr[i]

2) Don't Include.

F (i, k) = F (i+1 , k)

(int) bool

```
subsetSumK ( int i, int k) {
if(i == n) {
    if(sum == 0) return True;
    else return False;
}
)
```

→ (1)

↓ (0)

{ 0 ← case1 = (subsetsumk (i+1, k - arr[i]);
{ 2 ← case2 = (subsetsumk (i+1, k);

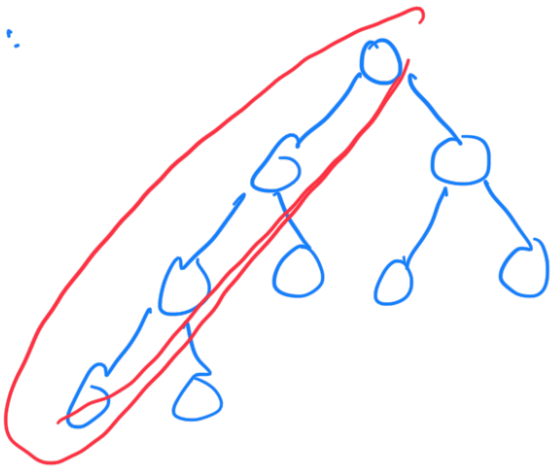
return case1 || case2;

```
bool SubsetSumk( i , k ) {
if(k == 0) return True;
if(i == n) return False;
```

$\frac{n(n+1)}{2}$

```
for(i=0; i<n; i++)
  for(j=i; j<n; j++)
```

T.C:



$O(2^n)$

$2^n \geq 10^6$

(1)

$n = 20$

$2^{20} \approx 10^6$

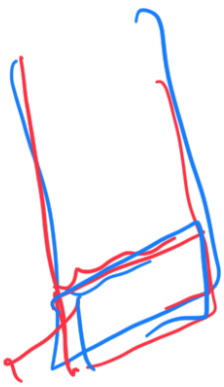
S.C: $O(n)$

```
fun(n){
{
  . . . .
  . . . .
  . . . .
return fun(n-1);
```

?

Tail Recursion

$A(n) \rightarrow f(n-1) \rightarrow f(n-2)$



a f(a-1)

X multi

```
bool subsetSumK( int i, int k) {
if(i == n) {
    if(sum == 0) return True;
    else return False;
}
case1 = subsetSumk(i+1, k-arr[i]);
case2 = subsetSumk(i+1, k);
return case1 || case2;
```

re[illegible]

}