Recursion

→ Recursion is not just a computer science topic
It corelates with maths as well.

Mathematically, recursion means a function calling itself.

$$f(x) = f(f(x'))$$

$f(a,b) = a \times f(a,b-1)$

↳ func^n definition

This function calculates a^b

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

Ex $f(2,3) \to 2^3 \to 8$

$f(2,4) \to 2^4 \to 16$

$2^4 = 2 \times 2^3$

In computer science, recursion means function calling itself.

```
function fun(x) {
    ...
    ...
    ...
    fun(x-1);
    ...
    ...
}
```

```
function gun(x) {
    console.log(x);
}

function fun(x) {
    ...
    gun(x-1)
    fun(x-1);
    ...
}
```

gun(x-1) → This is not recursion

fun(x-1); → This is recursion

Example-1

Factorial

factorial of any value n ⇒ $n \times (n-1) \times (n-2) \cdots \times 2 \times 1$

↳ n!

$5! = 5 \times 4 \times 3 \times 2 \times 1 \rightarrow 120$

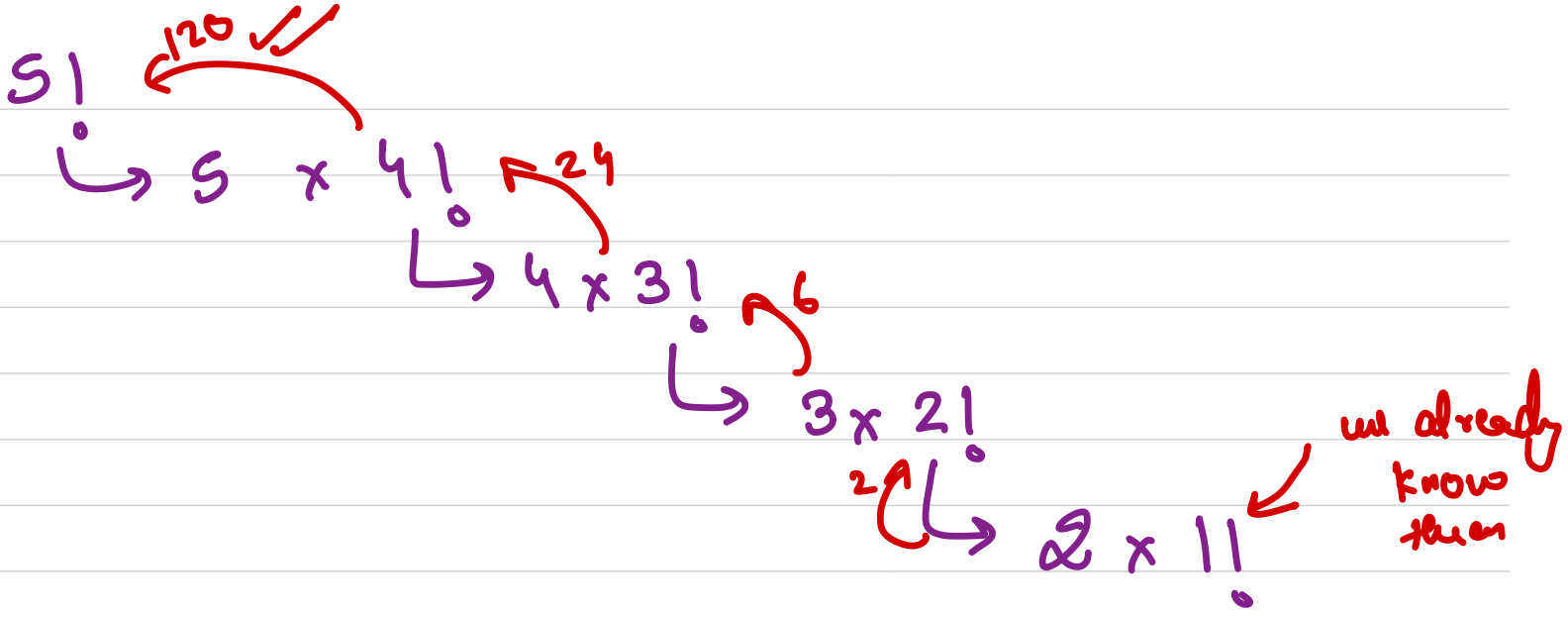
$4! = 4 \times 3 \times 2 \times 1 \rightarrow 24$

$3! = 3 \times 2 \times 1 \rightarrow 6$

$f(n) = n \times f(n-1)$

this function returns $n!$

The mathematical representation of a recursive function is called Recurrence Relation



What is $1!$? $\Rightarrow 1$

Recursion → In recursion, a function calls itself, where it tries to solve a smaller problem and then calc value of a bigger problem

PMI (principal of Mathematical Induction)

Q → What is the Sum of first N natural numbers ??

Ans → $\frac{n \times (n+1)}{2}$

3 step process

1) Base Case → the smallest value for which we can directly verify the ans.

2) Assumption

3) Verification / self task

for $n=1$ $\longrightarrow$ $\frac{1\times(1+1)}{2}$ $\longrightarrow$ is correct

↳ Base case

for some $n=k$ $\longrightarrow$ the formula is correct

i.e. $\frac{k\times(k+1)}{2}$

Let's prove ourselves that formula is correct for $n = k+1$

$$1 + 2 + 3 \text{ ------- } k + k+1$$

$$\rightarrow \frac{k \times (k+1)}{2} + (k+1)$$

$$\frac{k(k+1) + 2(k+1)}{2} \rightarrow \frac{(k+1)(k+2)}{2}$$

By formula → $n=(k+1)$

$$\frac{n\times(n+1)}{2} \rightarrow \frac{(k+1)(k+1+1)}{2}$$

$$= \frac{(k+1)(k+2)}{2}$$

Components Of Recursion

1) Base Case
2) Assumption
3) Self work

Write a recursive sol^n for factorial of n

We will write a func^n

$f(n)$

↓

returns n!

Base Case → $(n==1)$ $f(1) \rightarrow 1$

```
if (n==1) return 1
```

Assumption → assume the function works fine for n-1

f(n-1) correctly calcs (n-1)!

Selfwork → f(n) = n x f(n-1)

Power function a^b

$f(a, b)$

↓

returns a^b

Base Case → if $(b == 0)$ then ans is 1

Assumption → assum func^n works fine for $b-1$

i.e. $f(a, b-1)$ is correct.

Self work → $a^b = a \times a^{b-1}$

$f(a,b) \longleftarrow a \times f(a,b-1)$

(correct)

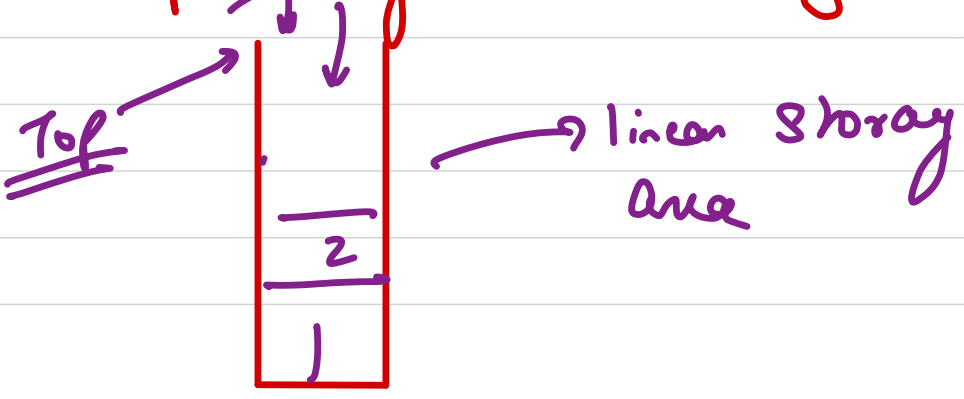
27^{51} $\quad 27 \times (27^{50}) =$

$27^{50} = y$

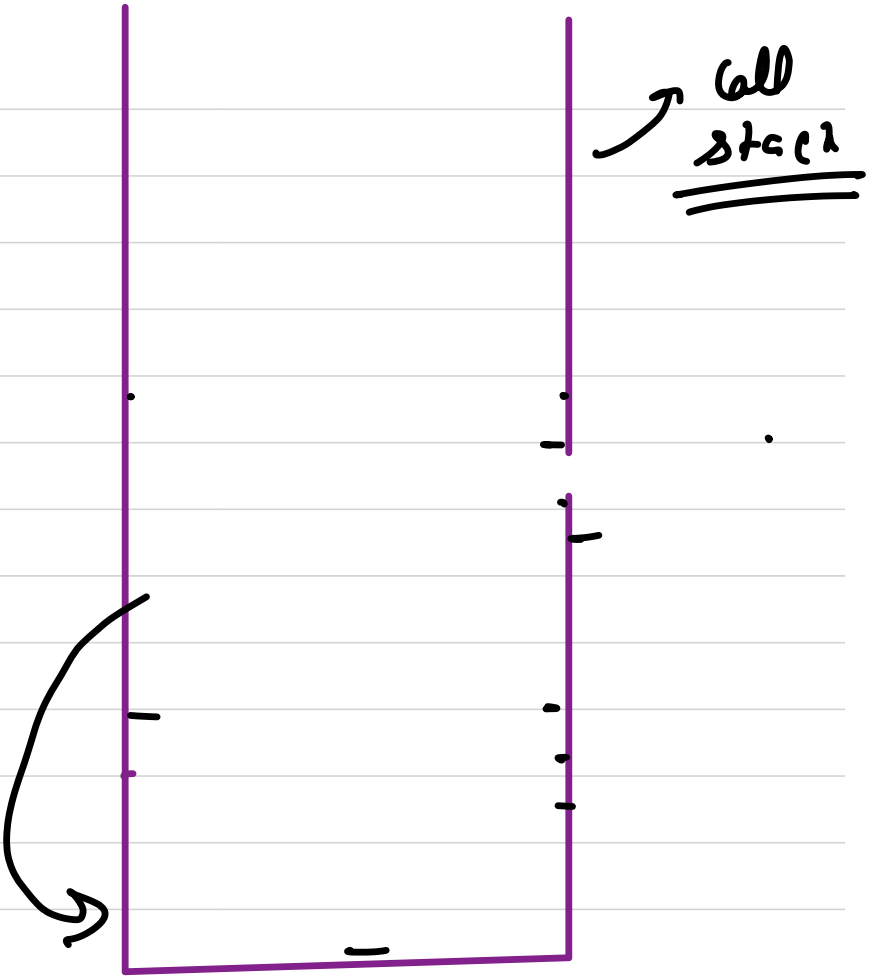
Q: What happens when we call a function??

In our memory, we have a lot of things going on.

One part of the memory is **call stack**



```
1 function fun(x)
2     let a = x + 10;
3     return a;
4 }
5 function gen(y)
6     let b = y + 20
7     let z = fun(b);
8     return z;
9 }
10 console.log(gen(10))
```



Whenever we call a new function it adds a new entry in the stack.

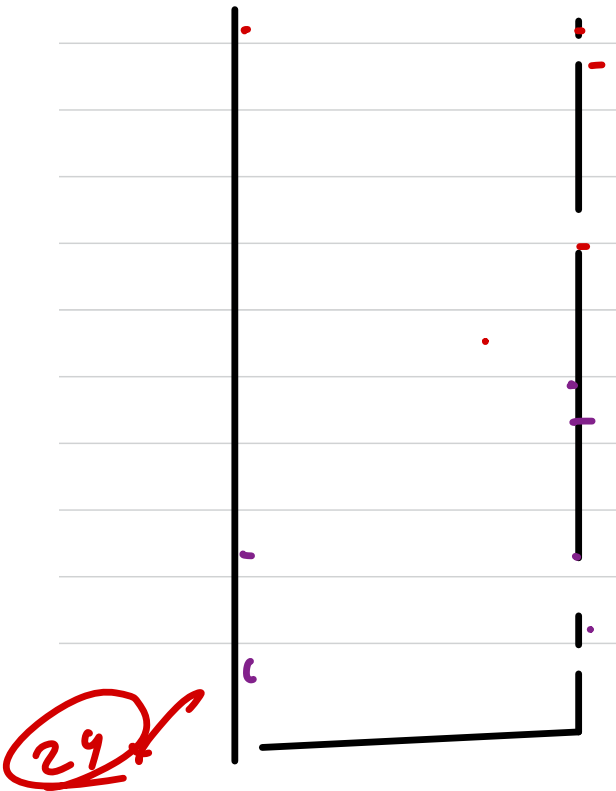
This new entry is called stack trace

Inside a trace we store all the variables and more data like current line of execution of function.

When we hit a return, the stack trace is removed & value is given back to the caller.

```
function fact(n) { // this function should calculate n!
    // Base case
    if(n == 1) return 1;
    let assume = fact(n-1); // that fact of n-1 is correct
    return n*assume;; // self work
}
```

fact (4)



Q. Given a value n, calc n^{th} fibonacci.

→ 0, 1, 1, 2, 3, 5, 8, 13, 21 ....

0^{th}	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	.....
fib	fib	fib	fib	fib	fib	fib	

n = 7

Ans = 13

Solve it recursively

n^{th} fib

$f(n)$

↓

returns n^{th} fib

Base Cases

$f(0) \rightarrow 0$

$f(1) \rightarrow 1$

} → stacks

Assumption → we assume $f(n-1)$ works

for & $f(n-2)$ works for

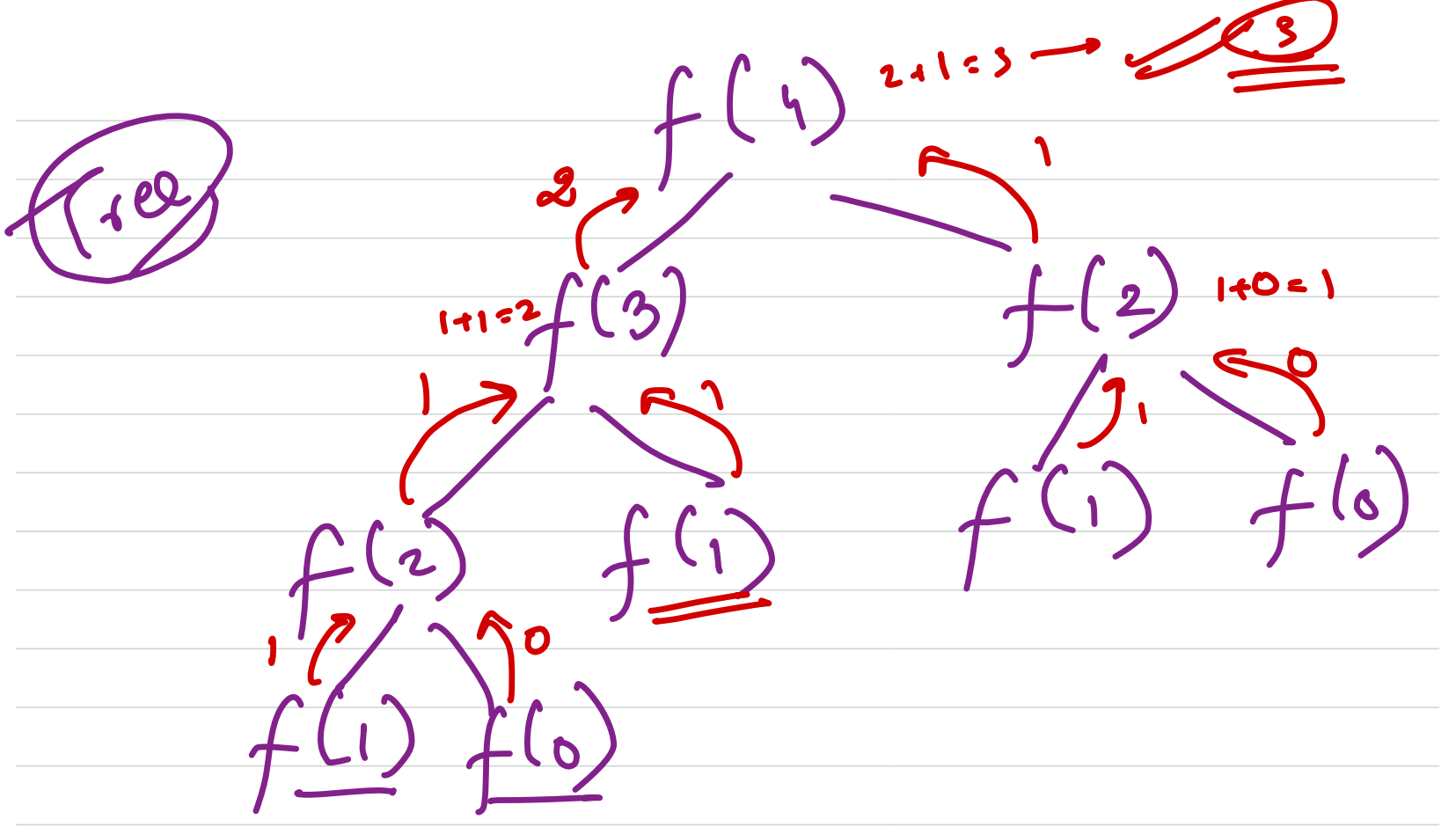
Self work → $f(n-1) + f(n-2)$

$f(n) = f(n-1) + f(n-2)$

```
function fib(n) {
    // base case
    if(n == 0) return 0;
    if(n == 1) return 1;
    // assumption
    let a = fib(n-1);
    let b = fib(n-2);
    // self work
    return a+b;
}

console.log(fib(3))
```

Recursion takes Extra space in memory


f(4)
2+1=3
3
2
1
f(3)
1+1=2
f(2)
1+0=1
1
1
1
0
f(2)
f(1)
f(1)
f(1)
f(0)
1
0
f(1)
f(0)

$$f(n) + f(n+1) = f(n+2)$$

$$f(n-1) + f(n) = f(n+1)$$

$$f(n-2) + f(n-1) = f(n)$$

Say $n+2 = x$

$$f(x-2) + f(x-1) = f(x)$$

Q→ Given a number n, print the first N natural no. in increasing order, recursively

n = 5

Base → n == 1 → console.log(1)

ans → 1
2
3
4
5

assum → if someone can give me the first 4 natural no.'s then we can do something

self → print 5

f(n) → f(n-1)

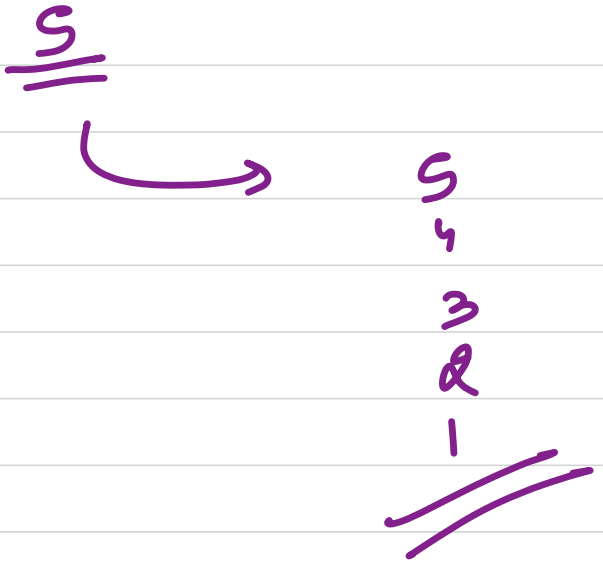
console.log(n)

which print

first n natural

no.s

assum f(n-1) works correctly



$f(n)$

n = 1

```
console.log(n)
f(n-1)
```

which prints first n natural no in decreasing order

f(n)

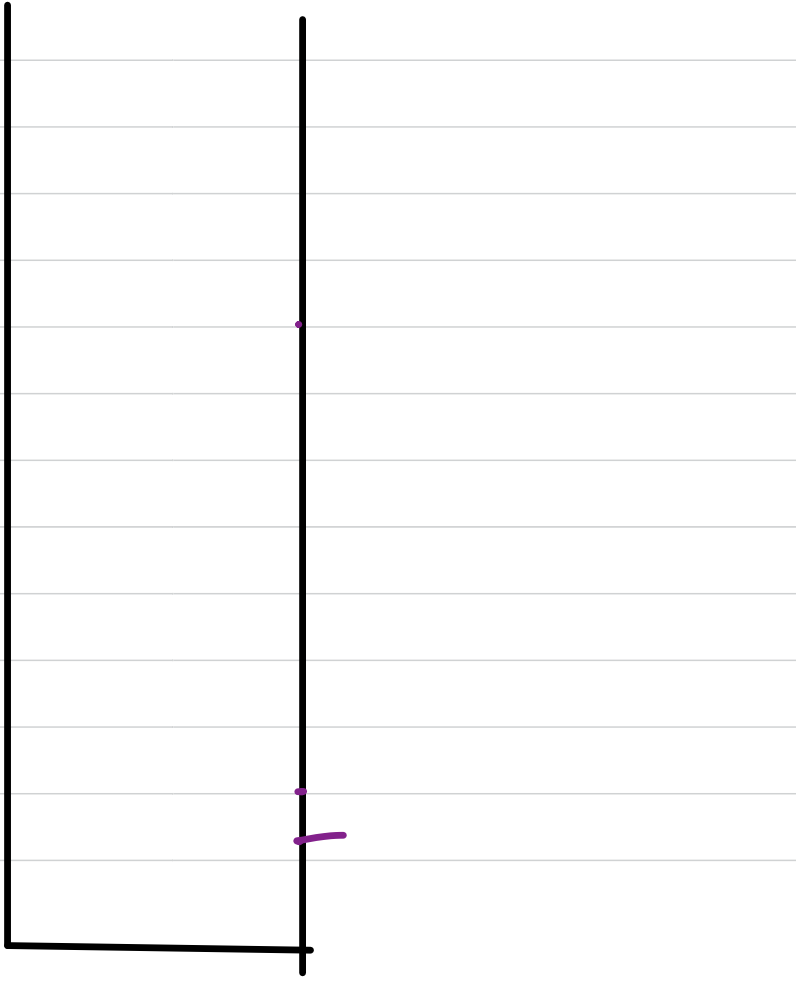
5
4
3
2
1

Console.log(n)

f(n-1) → assume it works fine

```
1   function print_inc(n) {
2       // base case
3       if(n == 1) {
4           console.log(1);
5           return;
6       }
7       print_inc(n-1);
8       console.log(n);
9   }
10
11  print_inc(3)
```

1
2
3



```
function print_dec(n) {
    // base case
    if(n == 1) {
        console.log(1);
        return;
    }
    console.log(n); // self task
    print_dec(n-1); // recursive assumption
}

print_dec(3)
```

3
2
1

Q. There are n friends, who want to go to a party. They can either go alone or go in a pair. Calc the no. of ways in which n friends can go to party. (Try Recursively)

Ex n = 3

ans → 4

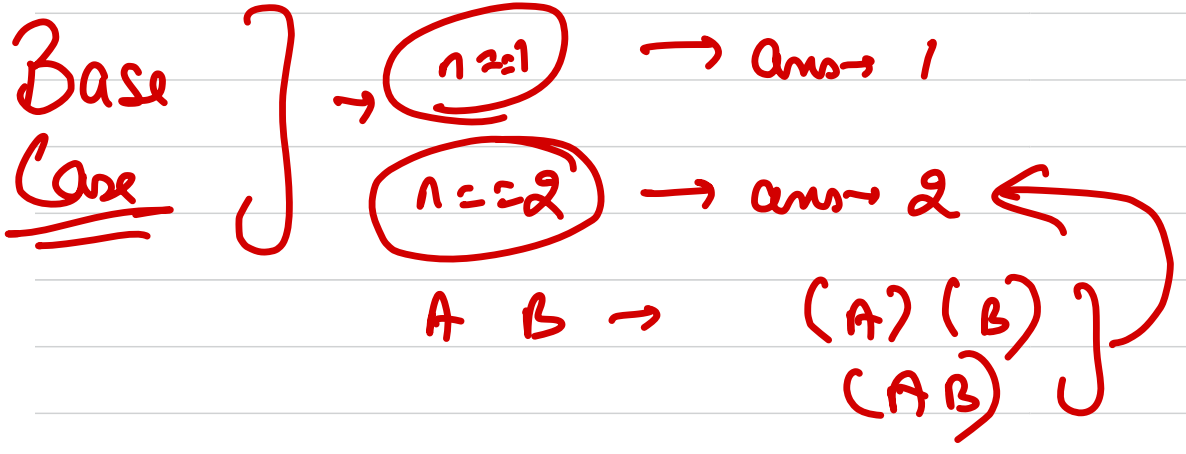
(A) (B) (C) →

(A B) (C) →

(A C) (B) →

(A) (B C) →

n=4 A B C D



n=4

A B C D

$f(n) = f(n-1) + (n-1) \times f(n-2)$

$f(n)$ ↓

returns the no. of ways in which n friends can go

$f(n-1)$: n^{th} friend said to go alone

$(n-1) \times f(n-2)$: n^{th} friend said to make a pair

n=4 A B C D → nth friend

D: alone / pair

what if D goes alone ??

(A) (B) (C) (D)

(AB) (C) (D)

(AC) (B) (D)

(BC) (A) (D)

What if n^{th} friend asked to make a pair ??

n^{th} friend

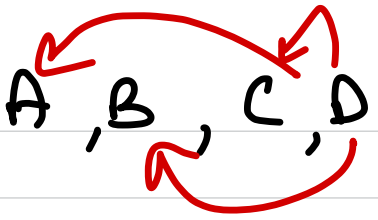
n=4 A B C D

In how many ways n^{th} friend can make a pair ?? → (n-1)

After making a pair how many elements left ?? → (n-2)

(n-1) x f(n-2)

3x2
=6



(DC) (A)(B)]
(DC) (AB)

(DB) (A)(C)]
(DB) (AC)

(AD) (B)(C)]
(AD) (BC)

6

$f(n-2) + f(n-2) + f(n-2)$

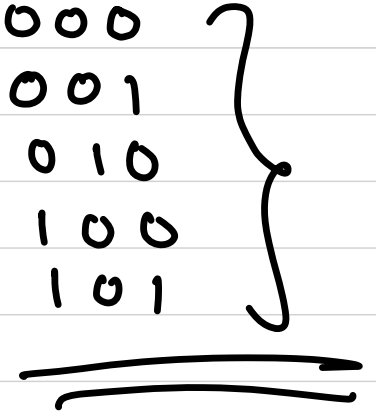
$(n-1) \times f(n-2)$

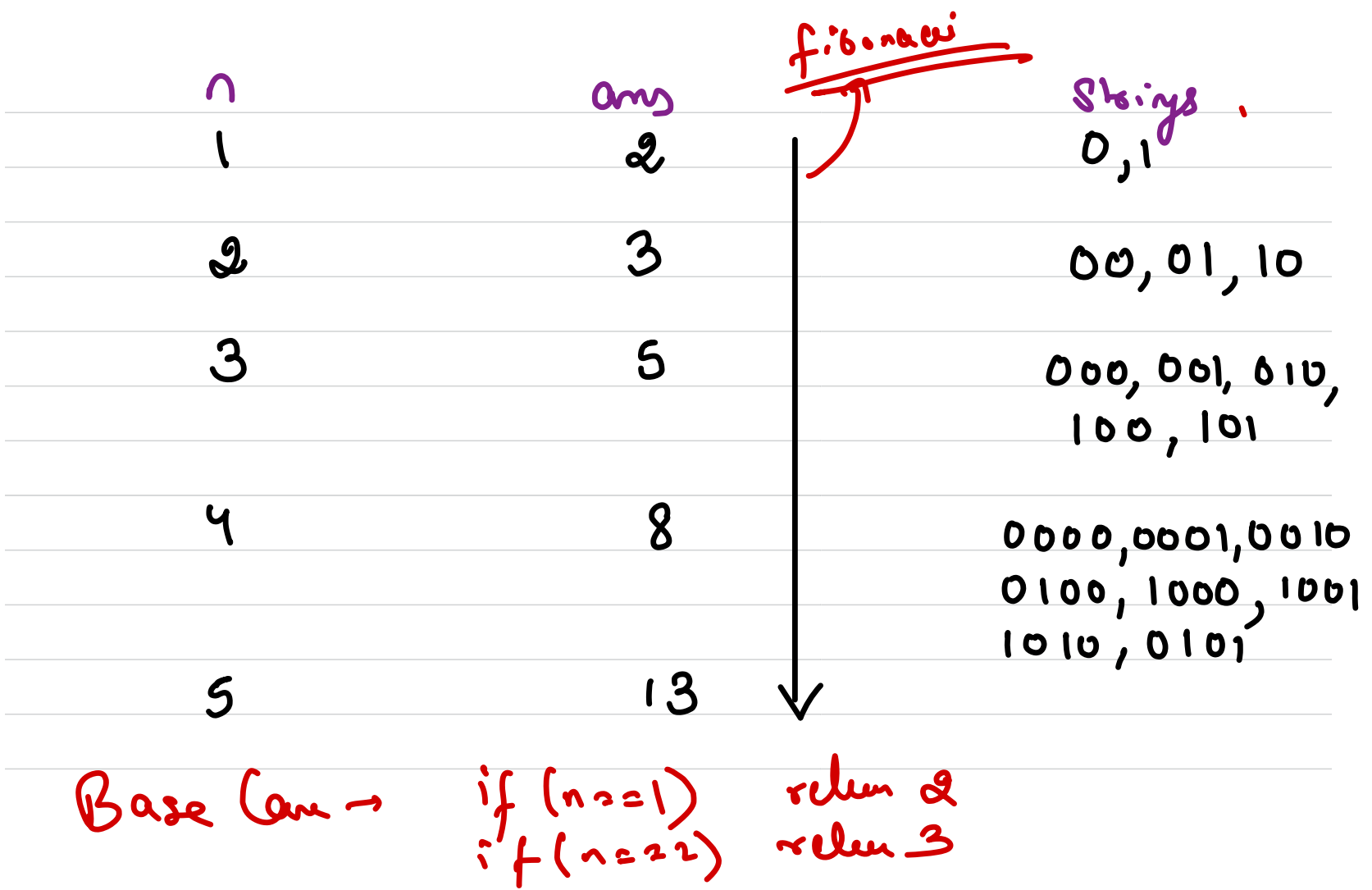
Q. Given a number n, calculate the no. of binary strings of length n with no consecutive one.

n = 3

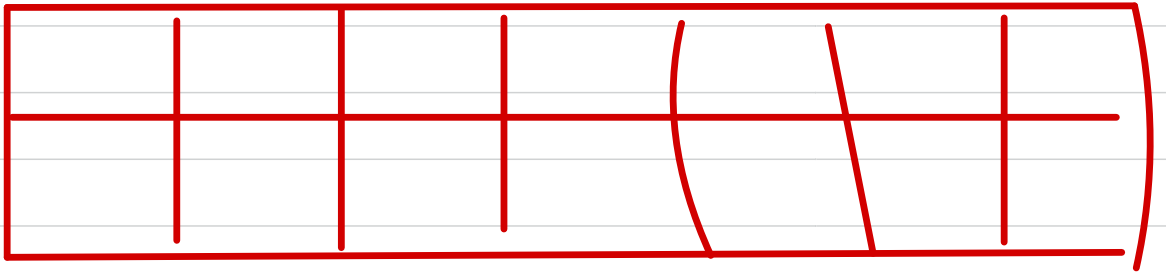
Ans → 5

000
001
010
100
101

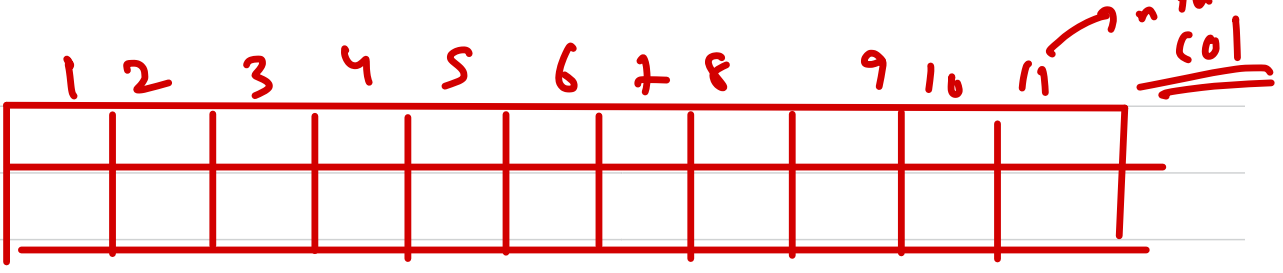



fibonacci
n
ans
Strings
1
2
0,1
2
3
00,01,10
3
5
000,001,010,
100,101
4
8
0000,0001,0010
0100,1000,1001
1010,0101
5
13
Base Case →
if (n==1)
if (n==2)
return 2
return 3

$$f(n) = f(n-1) + f(n-2)$$


2x1


2x1



2x1

On the n^{th} column you can put a tile either vertically or horizontally

$f(n) = f(n-1) + f(n-2)$

$f(n)$: return the no. of ways to put tiles on $2 \times n$ board

$f(n-1)$: n^{th} board we put a vertical tile

$f(n-2)$: n^{th} board with $(n-1)^{th}$ board filled with horizontal tile

Base Case

$n == 1 \rightarrow 1$

$n == 2 \rightarrow 2$


2
3
1

Q) Given an array, print all subsets of the array.

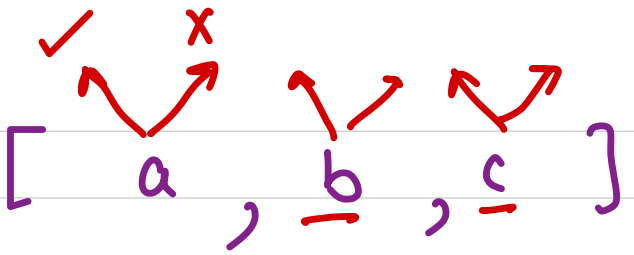
→ [1, 2, 3]

1
2
3
1 2
1 3
2 3
1 2 3
empty

If there is a set of length n,
how many subsets are possible ??

2^n

Permutation Combination

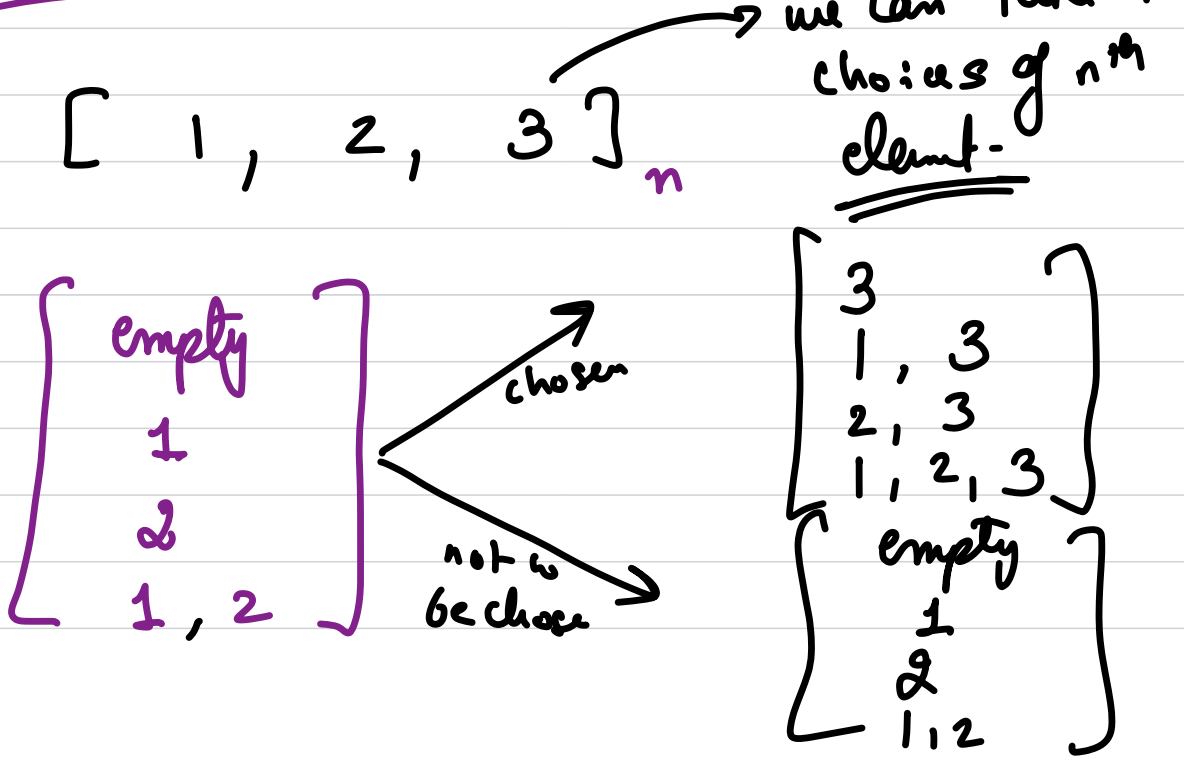


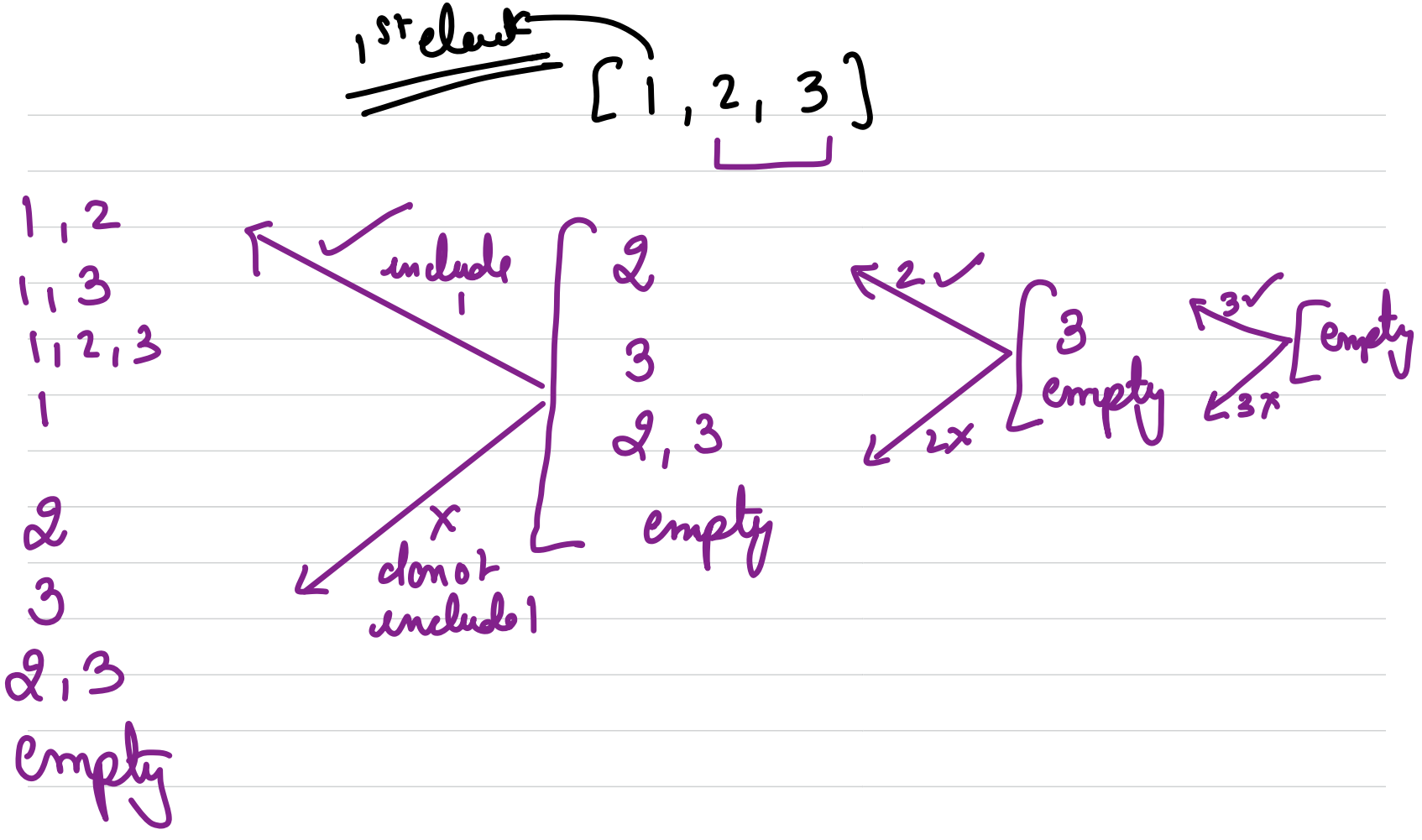
$2 \times 2 \times 2 \cdots$

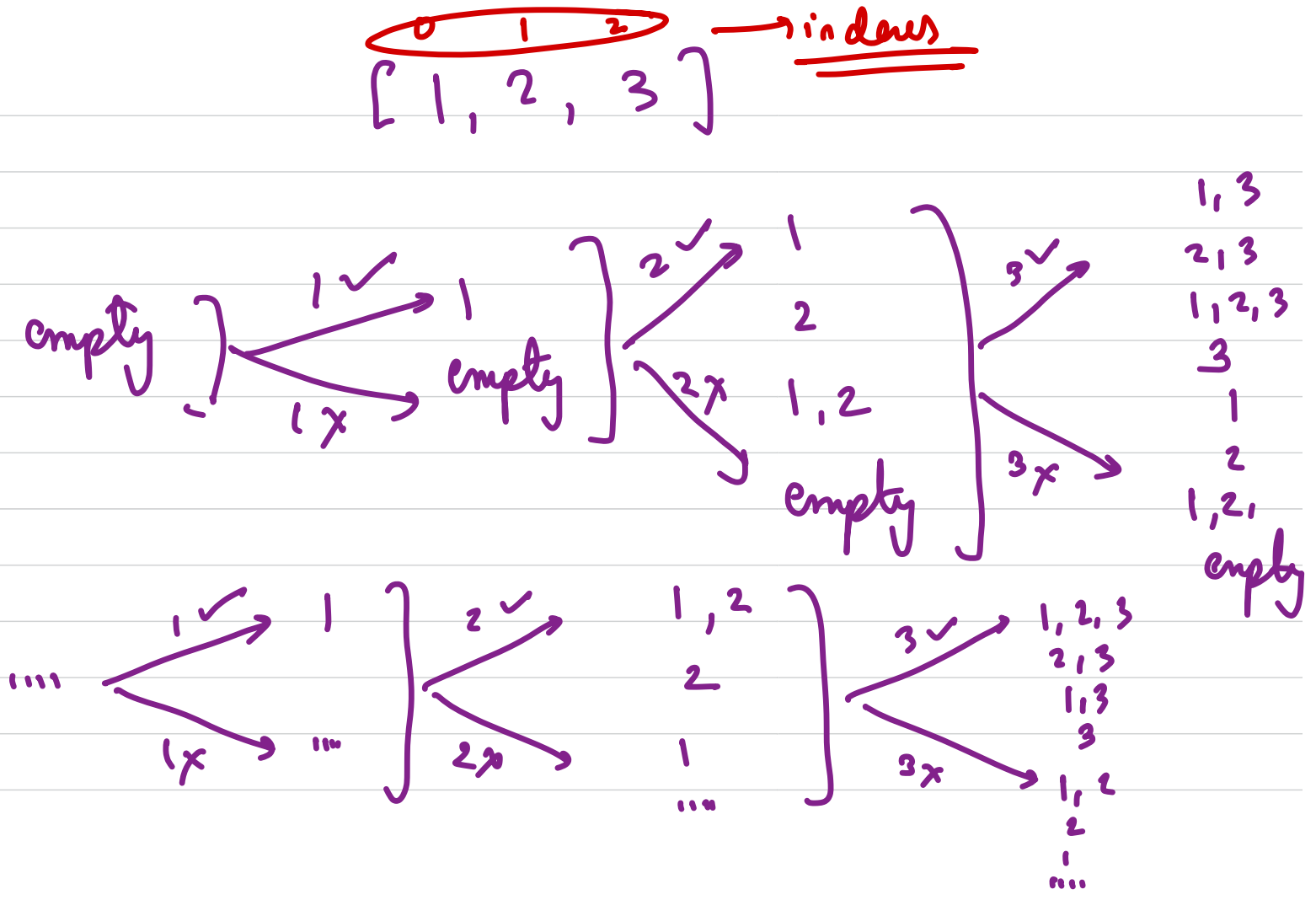
2^n

✗	✓	✓	→ b c
✓	✗	✓	→ a c
✗	✓	✗	→ b
✓	✓	✓	→ a b c
✗	✗	✗	→ empty
✓	✓	✗	→ a b
✗	✗	✓	→ c

We can use the same formula to build a recursive solution.




1st elem
[1, 2, 3]
include 1
x
donot include 1
2
3
2, 3
empty
2
2x
3
empty
3
3x
empty
1, 2
1, 3
1, 2, 3
1
2
3
2, 3
empty


0 1 2
indexes
[1, 2, 3]
empty
1 ✓
1 ✗
1
empty
2 ✓
2 ✗
1
2
1,2
empty
3 ✓
3 ✗
1, 3
2, 3
1, 2, 3
3
1
2
1, 2,
empty
.....
1 ✓
1 ✗
1
....
2 ✓
2 ✗
1, 2
2
1
....
3 ✓
3 ✗
1, 2, 3
2, 3
1, 3
3
1, 2
2
1
....

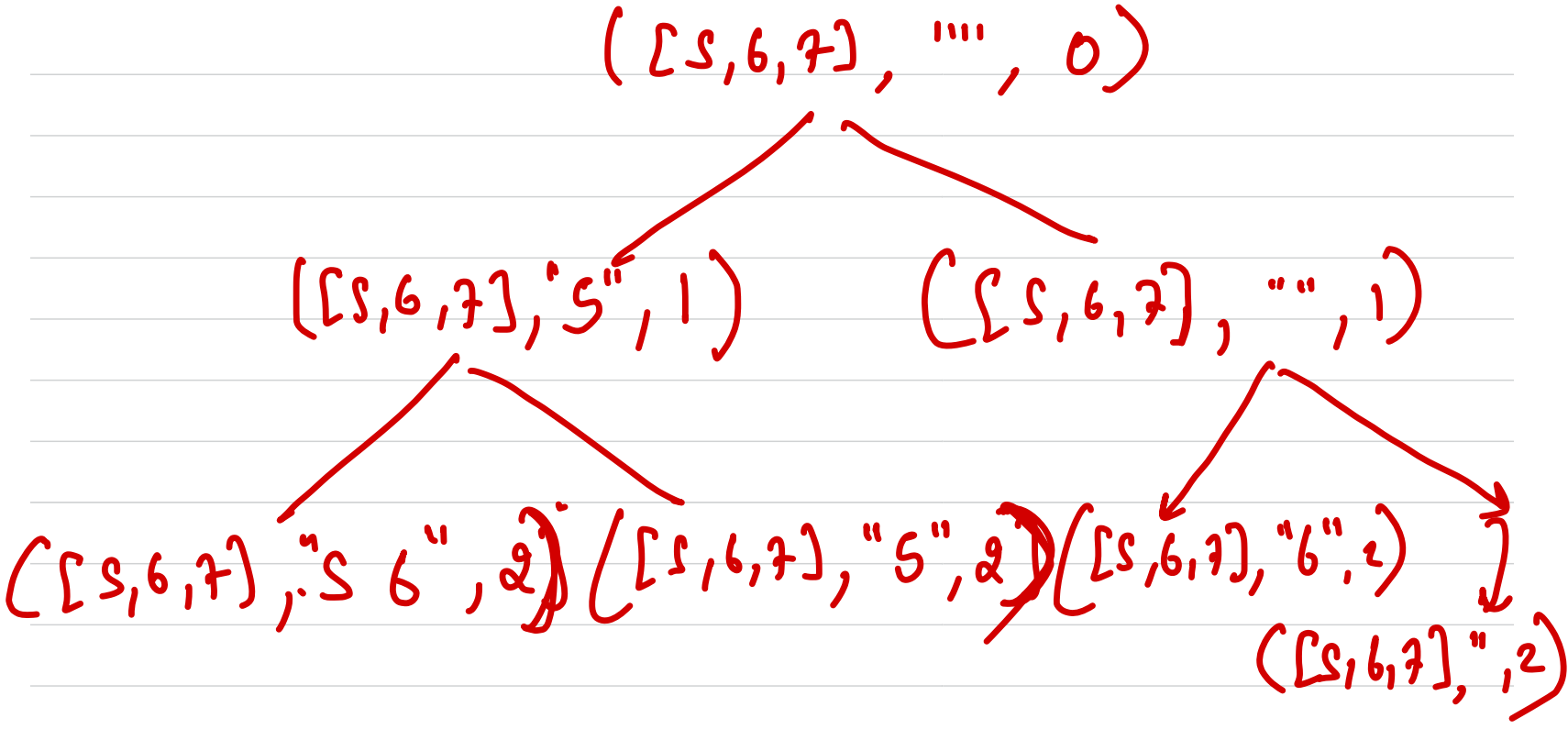
$f(arr, res, i) =$ $f(arr, res + arr[0], i+1)$ ← append

$f(arr, res, i+1)$

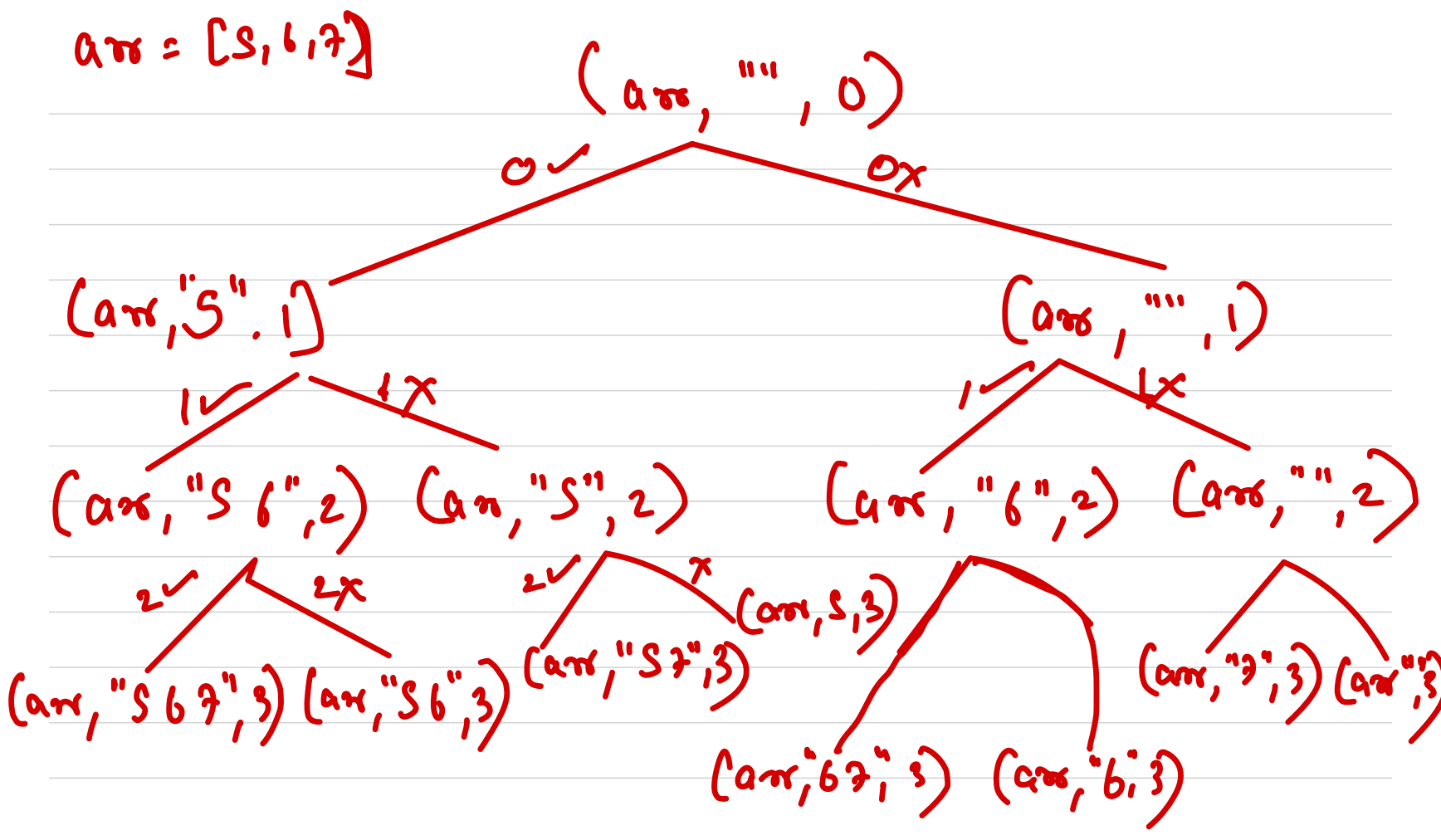
↳ this function prints all the subset of arr.

```
"3" + 7
"3 7"
```

Console.log(f([1,2,3], ""))


([5,6,7], "", 0)
([5,6,7], "5", 1)
([5,6,7], "", 1)
([5,6,7], "5 6", 2)
([5,6,7], "5", 2)
([5,6,7], "6", 2)
([5,6,7], "", 2)

arr = [5, 6, 7]


(arr, "", 0)
O✓
O✗
(arr, "5", 1)
(arr, "", 1)
1✓
1✗
1✓
1✗
(arr, "5 6", 2)
(arr, "5", 2)
(arr, "6", 2)
(arr, "", 2)
2✓
2✗
2✓
✗
(arr, "5 6 7", 3)
(arr, "5 6", 3)
(arr, "5 7", 3)
(arr, 5, 3)
(arr, "6 7", 3)
(arr, "6", 3)
(arr, "7", 3)

Q: Given an array, check if it is Sorted or not ?? (Recursively)

Sorted means arranged in ascendy order or not

[1, 2, 3] → True

[5, 9, 2] → false

Try recursively

Q) Given a number n, you can do any 3 operations on it any number of times

1) if n is divisible by 2, then divide it by 2
2) if n is divisible by 3, then divide it by 3
3) if n is greater than 1, then subtract 1 from it

Calc the minimum no. of operations required to reduce n to 1.

Ex n = 9 ans → 2

Say,

$f(n)$

↓

returns the min

no. of ops reqd to

reduce $n \rightarrow 1$ with

our given ops

$10 \xrightarrow{/2} 5 \xrightarrow{-1} 4 \xrightarrow{/2} 2 \xrightarrow{/2} 1 \quad \Rightarrow 4$

$10 \xrightarrow{-1} 9 \xrightarrow{/3} 3 \xrightarrow{/3} 1 \quad \Rightarrow 3$

Wrong Ans

Instead of reducing no. with any one op.
Try out all the possibilities.

Say,

$f(n)$ = $1 + \min \begin{cases} f(n/3) \\ f(n/2) \\ f(n-1) \end{cases}$

↓

returns the min

no. of ops reqd to

reduce $n \rightarrow 1$ with

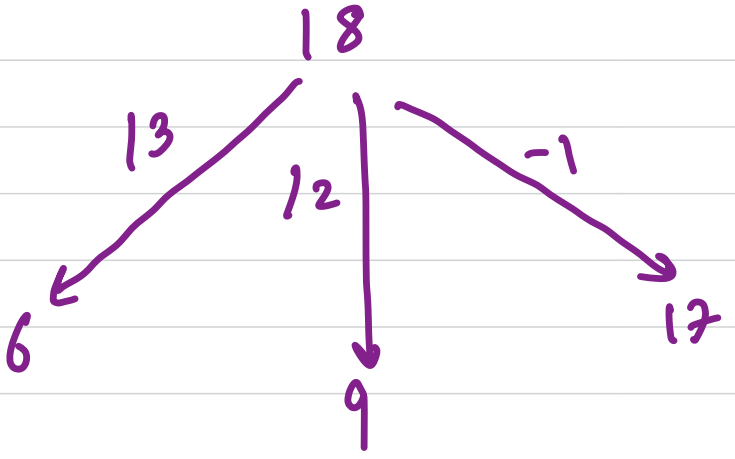
our given ops

Base Case

$n == 1 \rightarrow 0$

$n == 2 \rightarrow 1$

$n == 3 \rightarrow 1$

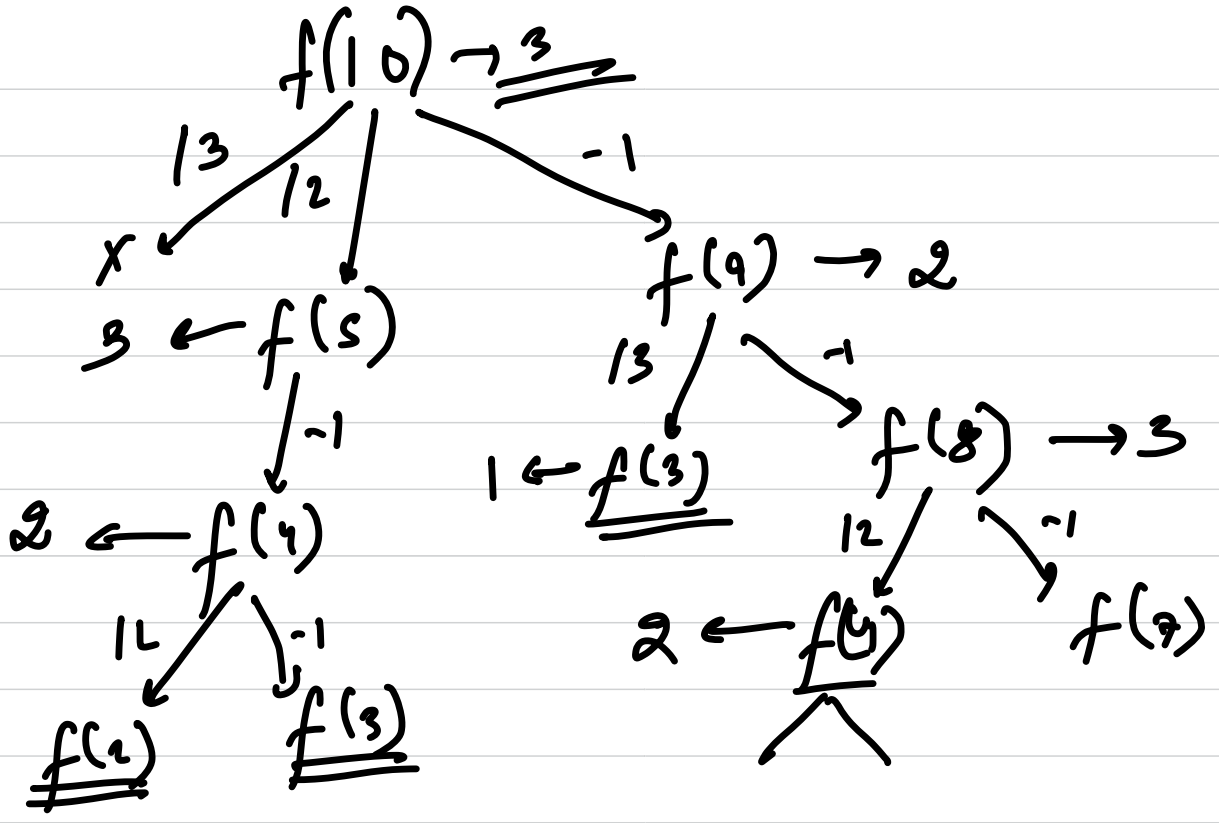


$6 \rightarrow 1 \rightarrow (x)$

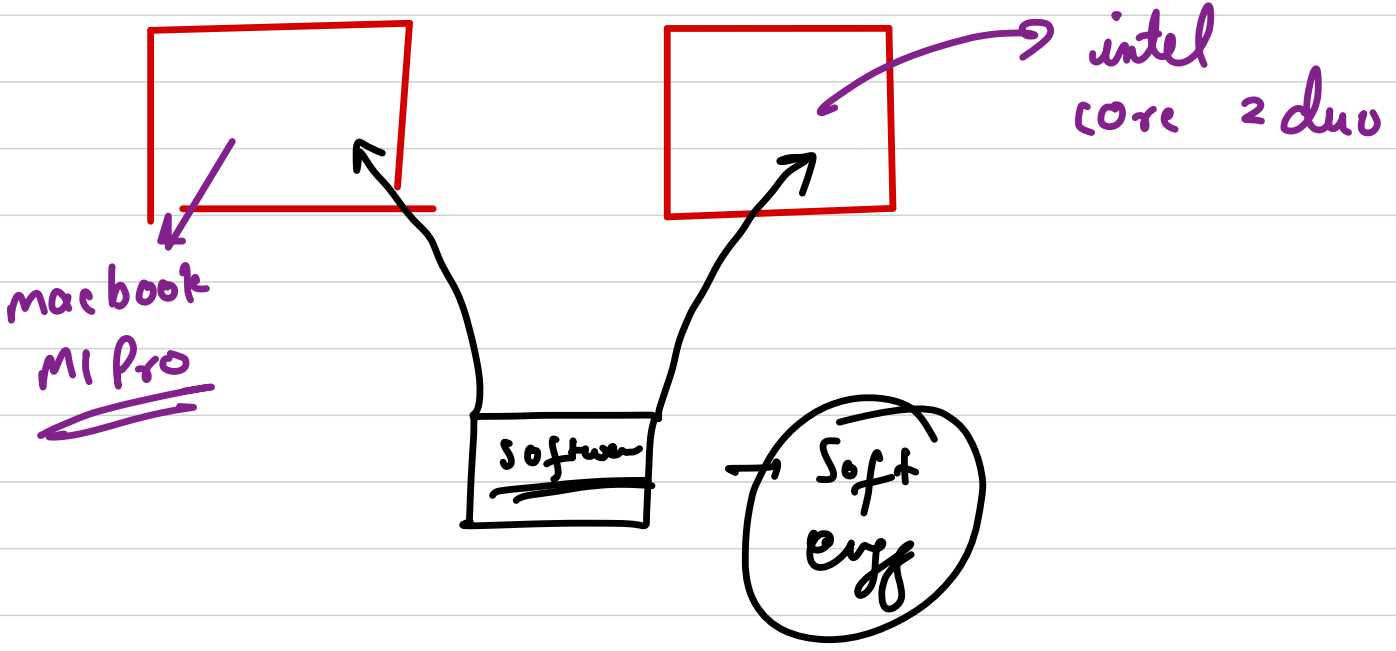
$9 \rightarrow 1 \rightarrow (y)$

$17 \rightarrow 1 \rightarrow (2)$

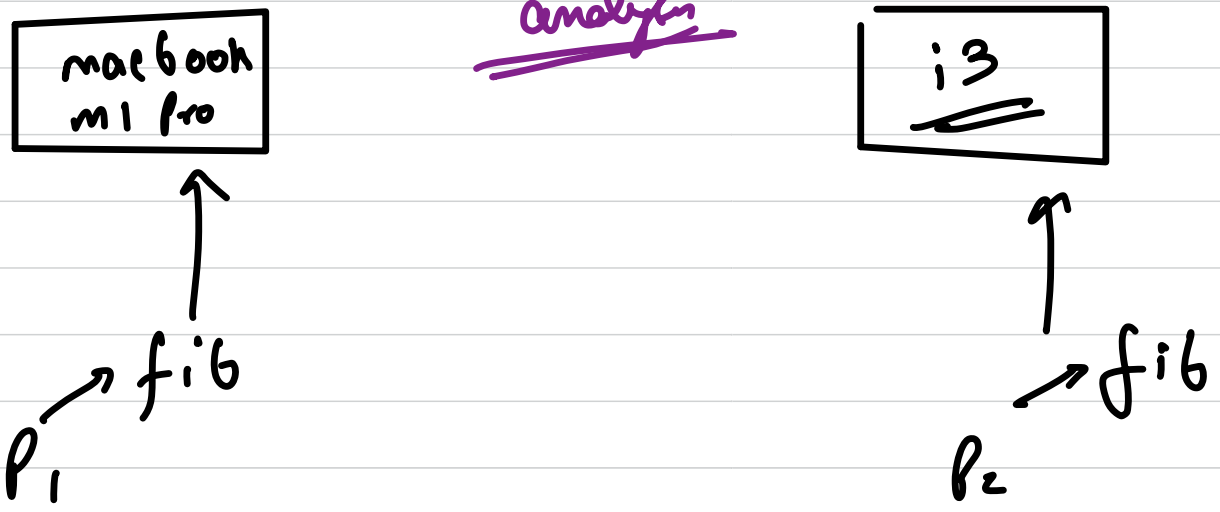
$1 + \min(x, y, 2)$


f(10) → 3
13
12
-1
x
3 ← f(5)
f(9) → 2
-1
13
-1
2 ← f(4)
1 ← f(3)
f(8) → 3
12
-1
1
-1
2 ← f(6)
f(7)
f(2)
f(3)

$$\min\left(\underbrace{+\infty}_{\text{↙}}, 3, 2\right)$$


intel
core 2 duo
macbook
M1 Pro
software
Soft
engg

Experimental analysis



In order to compare algorithms we need a mechanism which is machine agnostic

Asymptotic Analysis

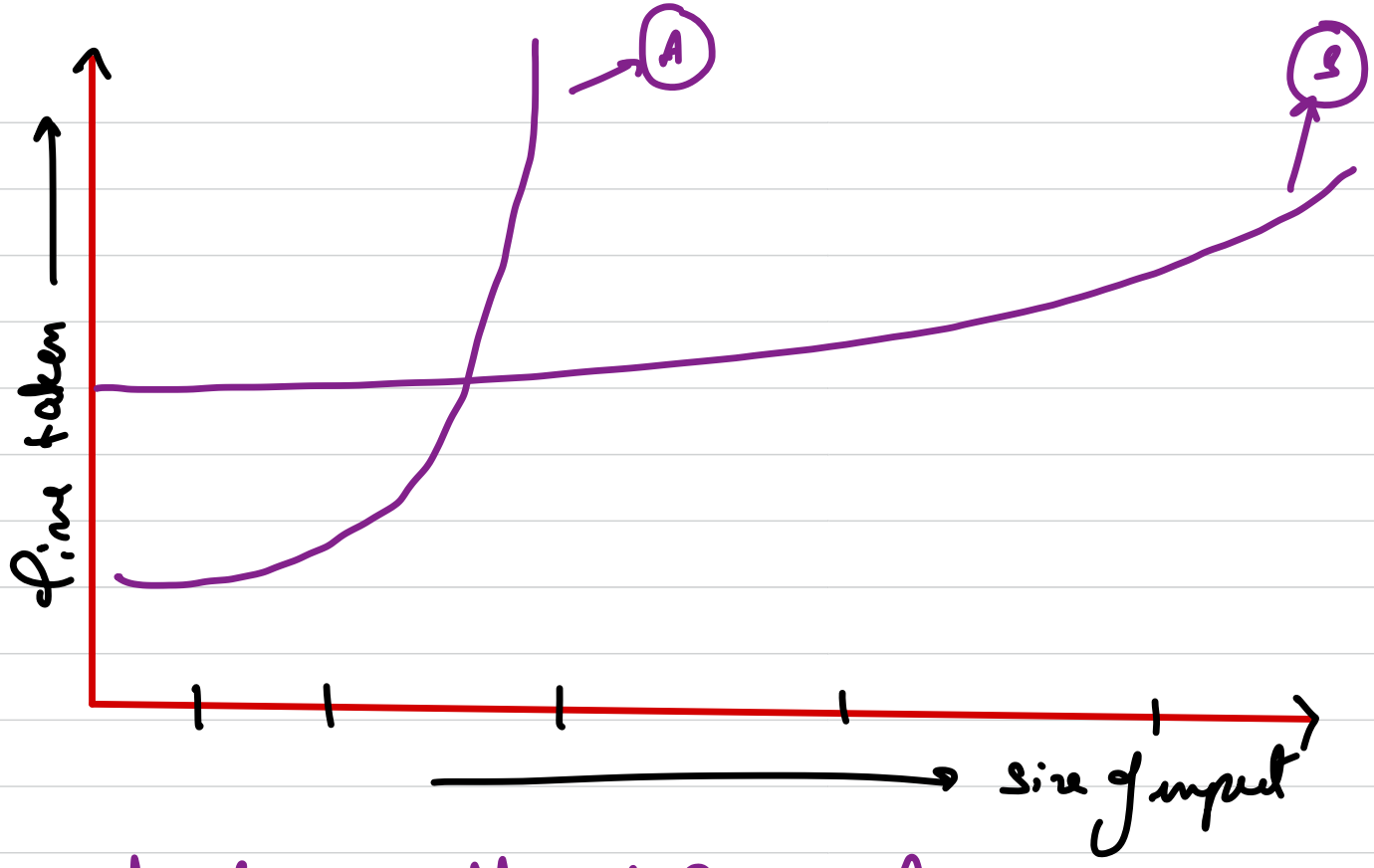
→ input size dependent

Asymptote

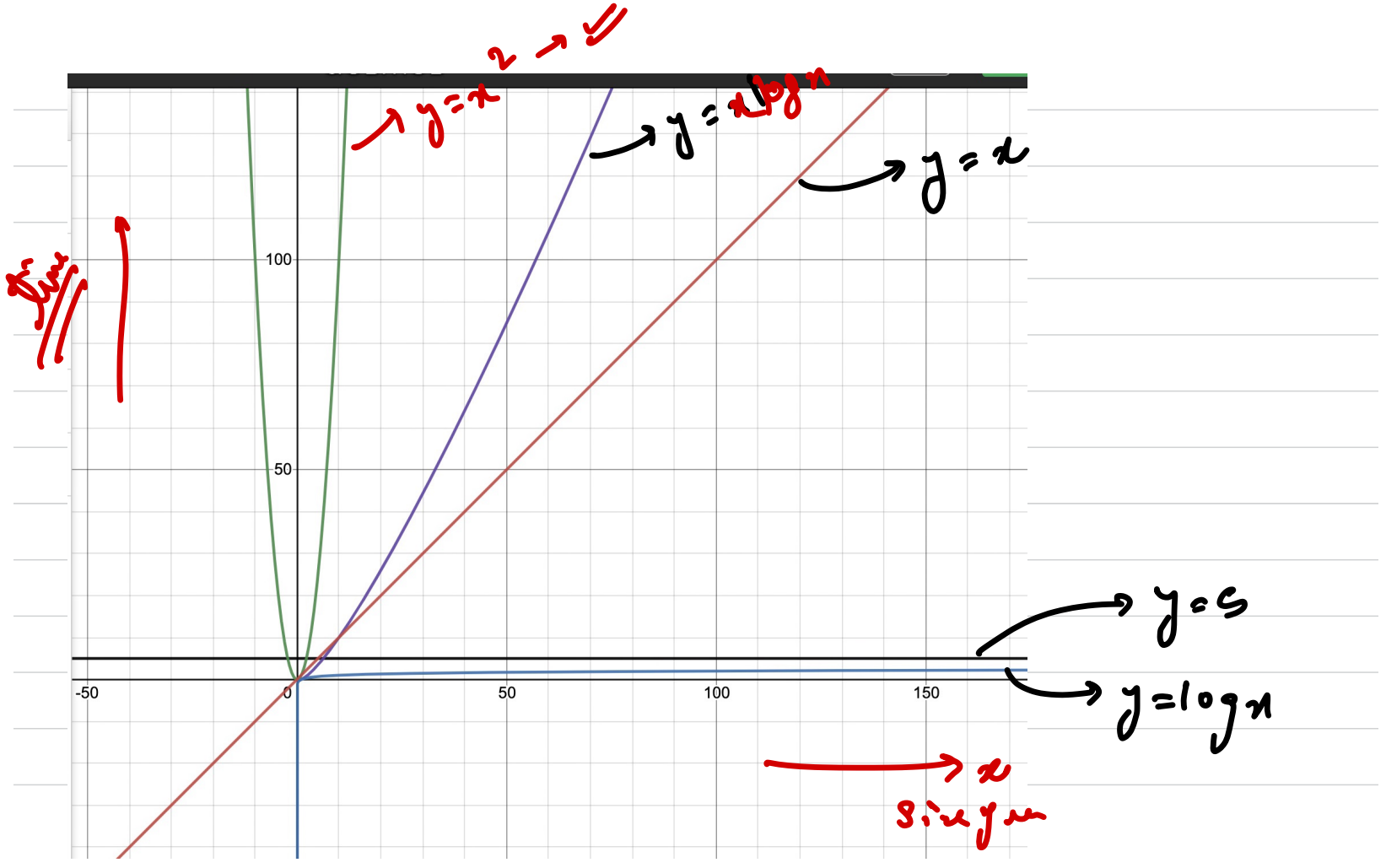
↳ On very big values of your input how the algo behaves is called asymptotic analysis

$2^3 \rightarrow 8$ ✓✓

$2^{10^6} \rightarrow$ ✓✓



The rate of growth of B is less


y=x^2
y=x log x
y=x
y=5
y=log x
100
50
-50
0
50
100
150
x
sin y=x

→ $y = x^2$ ⟶ $\frac{dy}{dx} \to 2x$

$y = x \log x$ ⟶ $\frac{dy}{dx} \to 1 + \log x$

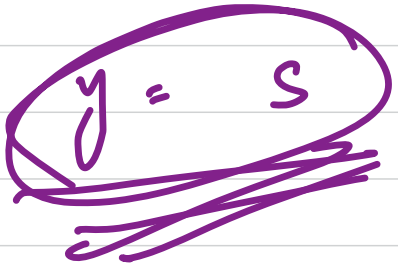
$y = x$ $\frac{dy}{dx} \to 1$

$y = \log x$ $\frac{dy}{dx} \to 1/x$

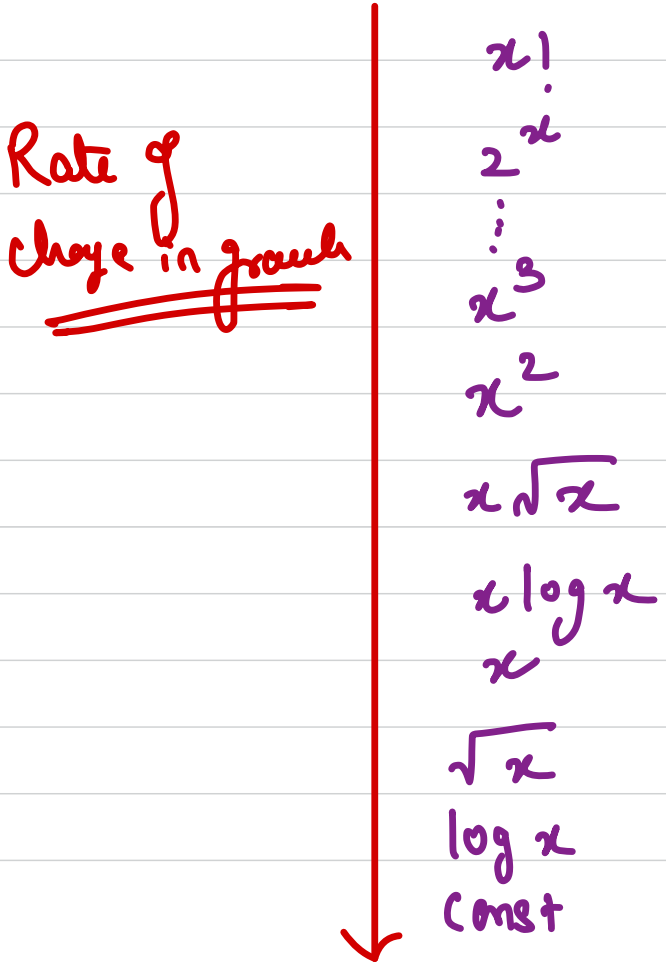
$y = 5$ $\frac{dy}{dx} \to 0$

$y = \log_2 x \rightarrow \log_2 10^6$ $\quad x = 10^6$

$\log_2 10^6 \rightarrow \approx 20$



$x = 10^6$


Rate of
Change in growth
x!
2^x
x^3
x^2
$x\sqrt{x}$
$x \log x$
x
$\sqrt{x}$
$\log x$
Const

```
for( i=1 ; i<=x ; i++) {
    console.log( i);
}
```

A ←

x

```
for (i=1 ; i<=x ; i++) {
    for (j=1 ; j<=x; j++) {
        console.log (i,j);
    }
}
```

→ B ←

x^2

1	x
2	x
3	x
4	x
⋮	⋮
x	x

$x + x + x \underbrace{\cdots\cdots}_{x \text{ times}} x = x^2$

Asymptotic analysis says that, for very high input value we can avoid lower degree terms and constants. We will judge rate of growth based on highest degree term only

$$y = 3x^3 + x^2 + 10x + 3$$

$x \to 10^9$

$$3 \times (10^9)^3 + (10^9)^2 + 10 \times 10^9 + 3$$

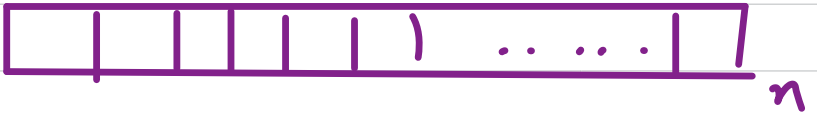
$\Rightarrow$

3 major Cases of algo analysis

1) Worst Case → priority highest

2) Avg Case

3) Best Case



$n \rightarrow 10^6$

element = x

Best Case → $\Omega(1)$

Worst Case → $O(n)$

Avg Case → $\Theta(c \times n) \Rightarrow \Theta(n)$

3 Cases → 3 notations

1) Worst Case → Big O notation

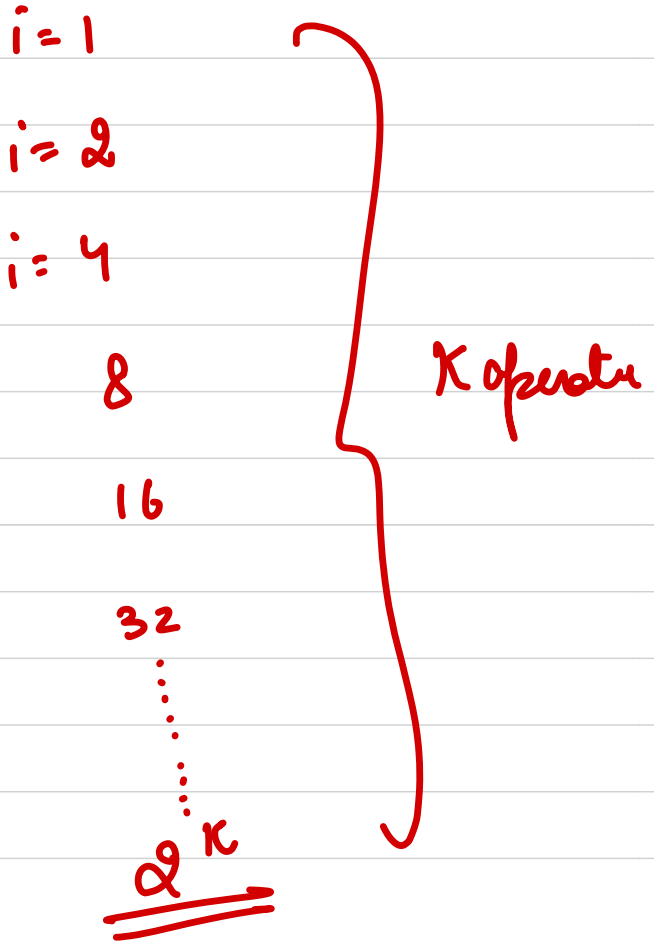
O → Oh → tightest upper bound

2) Avg Case → Big Θ notation

Θ → theta → avg tight

3) Best Case → Big Ω notation

Ω → omega → tightest lower bound



$k \approx \log_2 n$

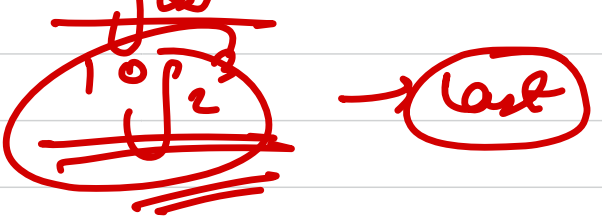
$2^k > n$

$\log_2 2^k > \log_2 n$

$k \log_2 2 > \log_2 n$

$k > \log_2 n$

$\log_3 x = \log_2 n$

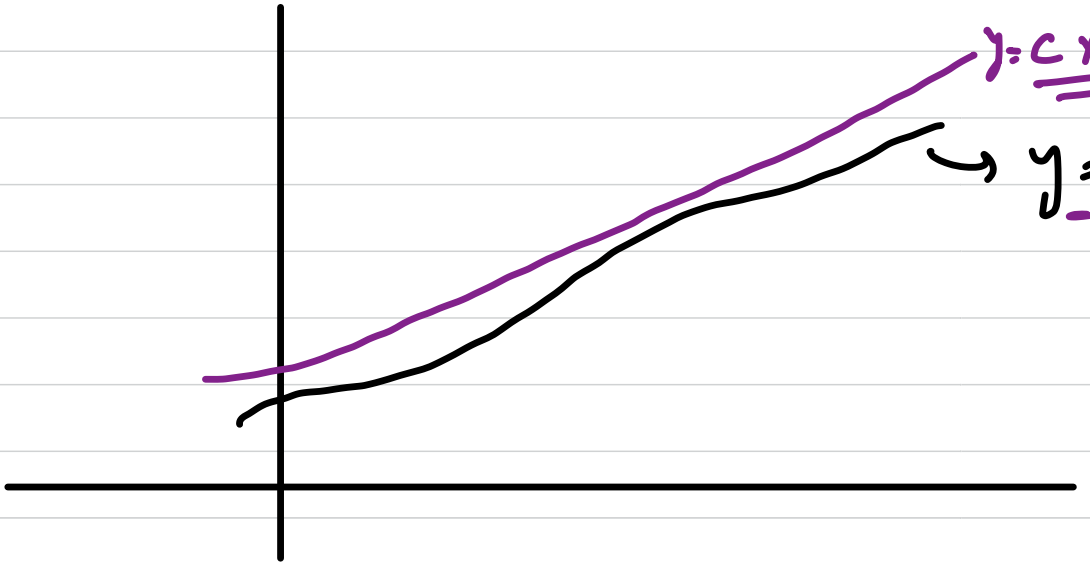


$i * i \leq n$

$i^2 \leq n$

$i \leq \sqrt{n}$

for (i = 1 ; i ≤ √n ; i + +)



c ≥ 3

$O(cx^2)$

$\approx O(x^2)$

Space Complexity

During the execution of algorithm apart from input and output space, how much space the algorithm took.

```
function sum(arr) {
    let ans = 0;
    for(let i=0; i<arr.length; i++) {
        ans += arr[i];
    }
    return ans;
}
```

recursion → call stack → will be created in space complexity

Doubt Session

* Dry run of subset
* Space Complexity
* Is array sorted

Subset ⇒ a subset of a set is a collection of elements extracted from the set.

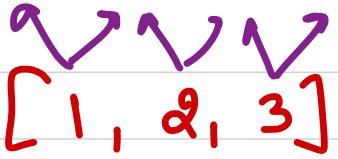
$[1, 2, 3]$ ⇒

2^n

$x_i \in S$

$\{1\}$
$\{2\}$
$\{3\}$
$\{1, 2\}$
$\{2, 3\}$
$\{1, 2, 3\}$
$\{1, 3\}$
$\{\phi\}$ ← empty

x



one choice → included

another choice → excluded

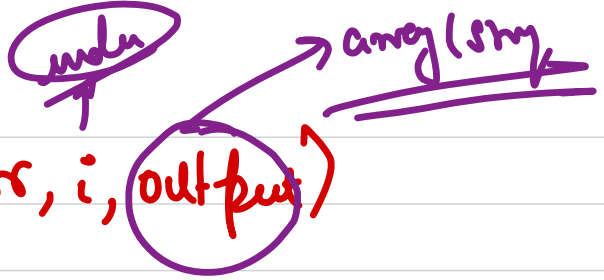
$2 \times 2 \times 2 \rightarrow 2^3$

[1, 2, 3]

2^n

✓	✓	✓	→ {1, 2, 3}
✗	✗	✗	→ {}
✓	✗	✗	→ {1}
✗	✓	✗	→ {2}
✗	✗	✓	→ {3}
✓	✓	✗	→ {1, 2}
✓	✗	✓	→ {1, 3}
✗	✓	✓	→ {2, 3}

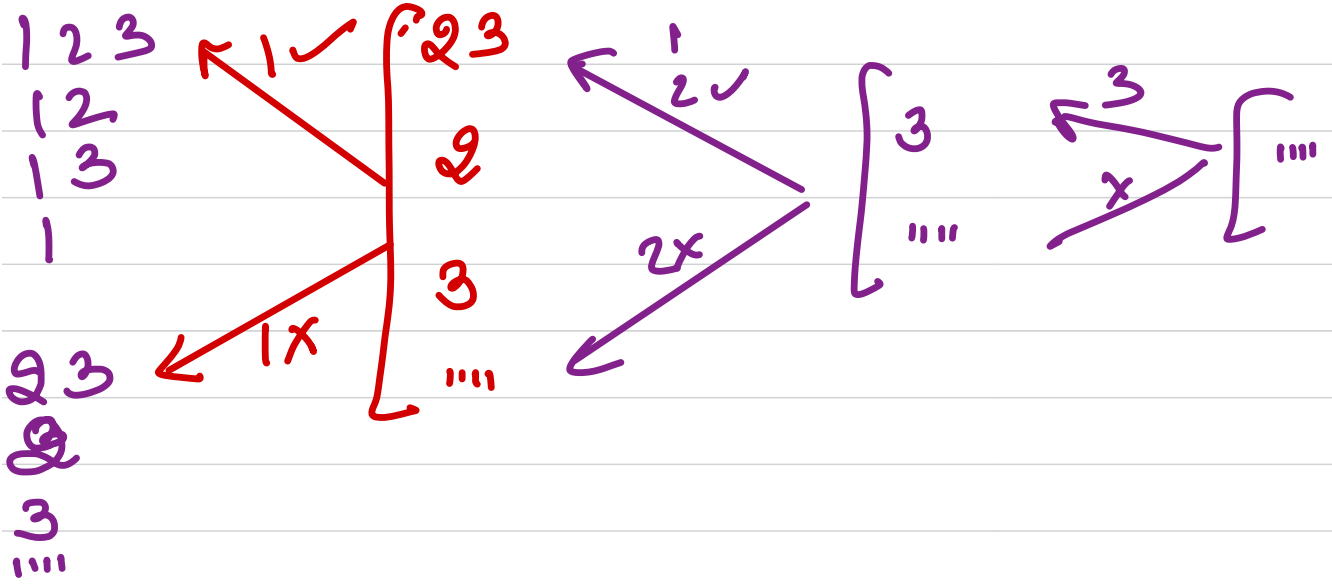
print

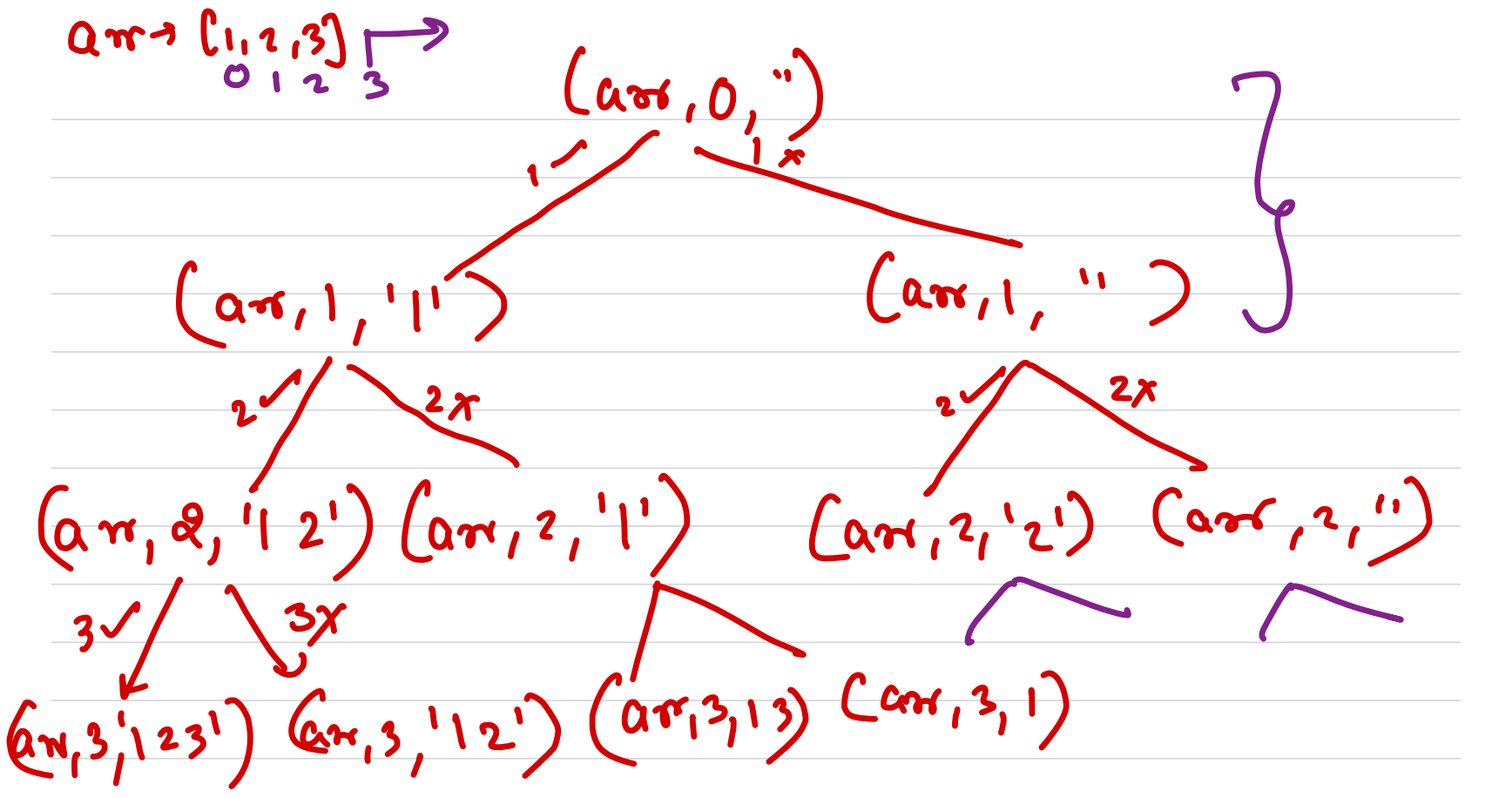


Prints the subset of arr starting from index i

f(arr, 0, "") → all

[1, 2, 3]




arr → [1,2,3]
0 1 2 3
(arr, 0, '')
1✓
1✗
(arr, 1, '1')
(arr, 1, '')
2✓
2✗
2✓
2✗
(arr, 2, '1 2')
(arr, 2, '1')
(arr, 2, '2')
(arr, 2, '')
3✓
3✗
(arr, 3, '123')
(arr, 3, '12')
(arr, 3, 13)
(arr, 3, 1)

```
function subset(arr, res, i) {
    if(i == arr.length) {
        console.log(res);
        return;
    }
    subset(arr, res + " " + arr[i], i+1);
    subset(arr, res, i+1);
}

subset([4, 9], "", 0);
```

Q → Recursively → arranged in asc

Check if array is sorted or not.

[1, 2, 3] → True

[3, 2, 1] → false

f(arr, idx)

2

function return whether array from index idx is sorted or not

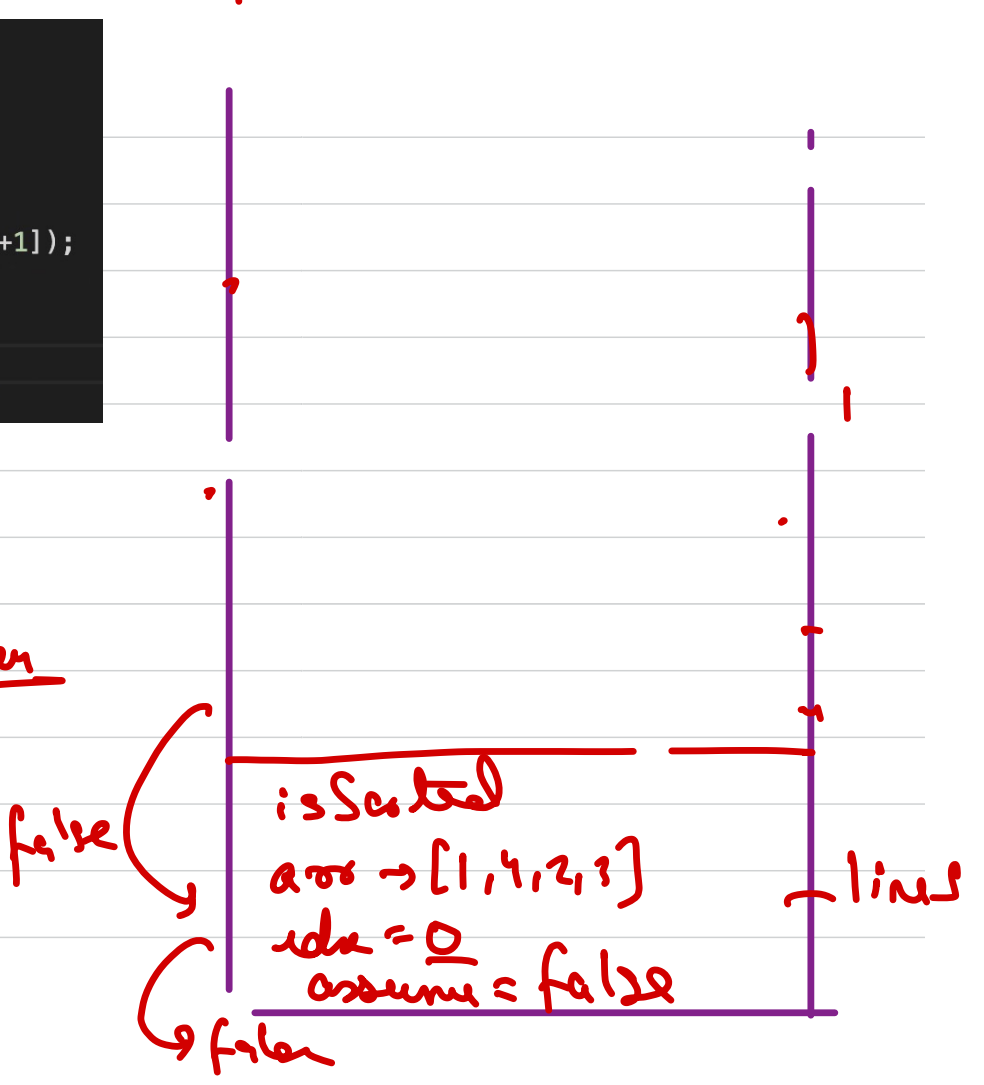
```
1  function isSorted(arr, idx) {
2      if(idx == arr.length - 1) {
3          return true;
4      }
5      let assume = isSorted(arr, idx+1);
6      return assume && (arr[idx] <= arr[idx+1]);
7  }
8
9  console.log(isSorted([1,12,3,4,5], 0));
```

arr → [1, 4, 2, 3]
0 1 2 3 → index

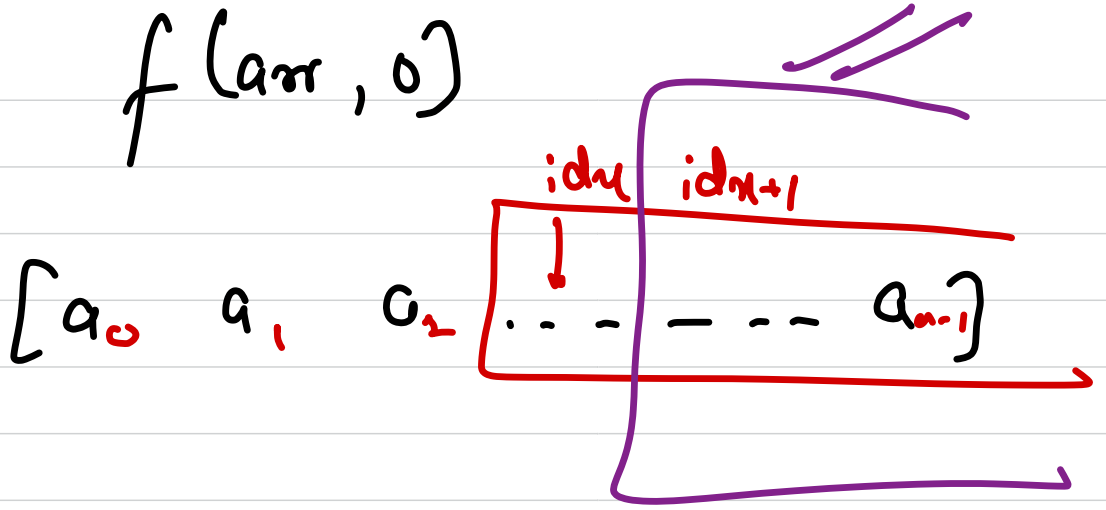
108

arr.length → 4

Space → O(n)



ans → f(arr, 0)



Base Case → if there is only single element → it is always sorted.

$f(arr, 0) = f(arr, 1)$ and $(arr[0] \leq arr[1])$

$f(arr, idx) = f(arr, idx+1)$ and $(arr[idx] \leq arr[idx+1])$

assume this works correctly

And :-

X	Y	X and Y
F	F	F
T	F	F
F	T	F
T	T	T

1, 2, 3, 4, 5

```
function f5(n) {
    for(let  i = n; i > 0; i/=2) {
        for(j = 0; j < i; j++) {
            console.log(i, j);
        }
    }
}
```

Total ops

$\rightarrow n + \frac{n}{2} + \frac{n}{4} \ldots\ldots$

$i = n \longrightarrow$ ops $\rightarrow n$

$i = \frac{n}{2} \longrightarrow$ ops $\rightarrow n/2$

$i = n/4 \longrightarrow$ ops $\rightarrow 1/4$

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \cdots\cdots\cdots\cdots\cdots\cdots 1$$

$$n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \cdots\cdots\cdots \frac{1}{n}\right) \Rightarrow 2n$$

What is the Sum of this Series $\Rightarrow 2$

geometric progression

$$S = \frac{a \times (1 - r^{n+1})}{1 - r} \Rightarrow \frac{a}{1 - r}$$

$$\frac{1}{1 - \frac{1}{2}} \rightarrow \left(2 - 1\right)/2 \Rightarrow 2$$

```
function f6(n) {
    let ans = 0;
    while(n > 0) {
        ans += n;
        n /= 2;
    }
    return ans;
}
```

k iterations

```
function f5(n) {
    for(let  i = n; i > 0; i/=2) {
        for(j = 0; j < i; j++) {
            console.log(i, j);
        }
    }
}
```

$n + \frac{n}{2} + \frac{n}{4} \ldots\ldots\ldots \rightarrow O(n)$

no. of iterations × 2 = no. of ops

$2 + 2 + 2 + 2 + 2 \ldots\ldots 2$

k times

```
function f6(n) {
    let ans = 0;
    while(n > 0) {
        ans += n;
        n /= 2;
    }
    return ans;
}
```

$2 + 2 + 2 + 2 \dots\dots 2$ (K iterations)

K → 2

no. of iteration

last value of n

$n \geq 1$

Total no. of ops = $2K$

$n \xrightarrow{1} \frac{n}{2} \xrightarrow{2} \frac{n}{2^2} \xrightarrow{3} \frac{n}{2^3} \xrightarrow{4} \frac{n}{2^4} \dots\dots \xrightarrow{K} \frac{n}{2^K}$ (last iteration)

$\frac{n}{2^K} \geq 1$

$$n \geq 2^k$$

Taking log both sides

$$\log_2 n \geq \log_2 2^k$$

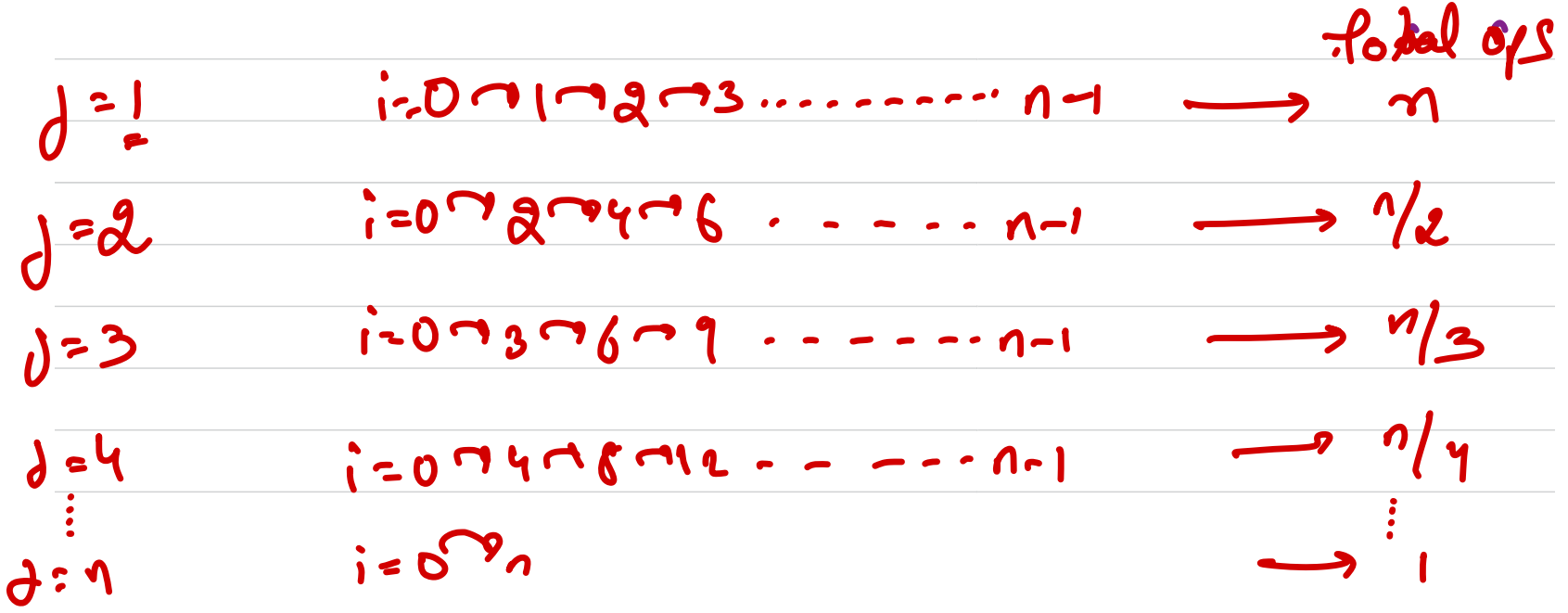
$$\log_2 n \geq k \log_2 2$$

$$\boxed{k \leq \log_2 n}$$

Total ops → $2k$ → $2\log_2 n$

→ $O(\log n)$

```
function f7(n) {
    for(let j = 1; j <= n; j++) {
        for(let i = 0; i < n; i = i + j) {
            console.log(i, j);
        }
    }
}
```



$$n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} \cdots\cdots\cdots n$$

$$n\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \cdots\cdots \frac{1}{n}\right)$$

Sum of the series $< ?$ $\longrightarrow$ $\log n$

$O(n \log n)$

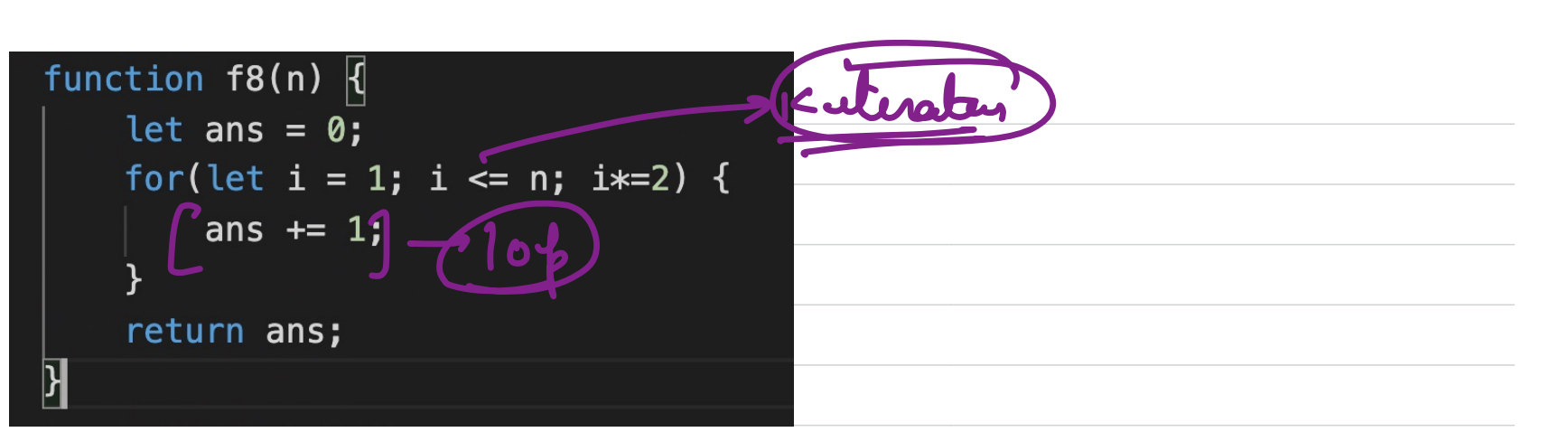
ans

$\sum_{k=1}^{n} \frac{1}{k}$ $\Rightarrow$ $\int_1^n \frac{1}{k}\,dk \rightarrow [\log k]_1^n$

$\log n - \log 1$

$\rightarrow \log n$

```
function f8(n) {
    let ans = 0;
    for(let i = 1; i <= n; i*=2) {
        ans += 1;
    }
    return ans;
}
```



Total iterations x 1 = Total ops

$i = 2^0$

$i = 2$

$i = 2^2$

$i = 2^3$

$i = 2^4$

$\vdots$

$i = 2^k$

$2^k \leq n$

Taking log both sides

$\log_2 2^k \leq \log_2 n$

$k \log_2 2 \leq \log_2 n$

$k \leq \log_2 n$

Time Comp $\Rightarrow O(\log n)$

time = # of operations in one function call × # of function calls

fact(n) → fact(n-1) → fact(n-2) → fact(n-3) ...

..... → fact(0)

fact(5) → fact(4) → fact(3) → fact(2) → fact(1) fact(0)

Time → O(1) x n

→ O(n)

Fib → #of ops in one func call → constant

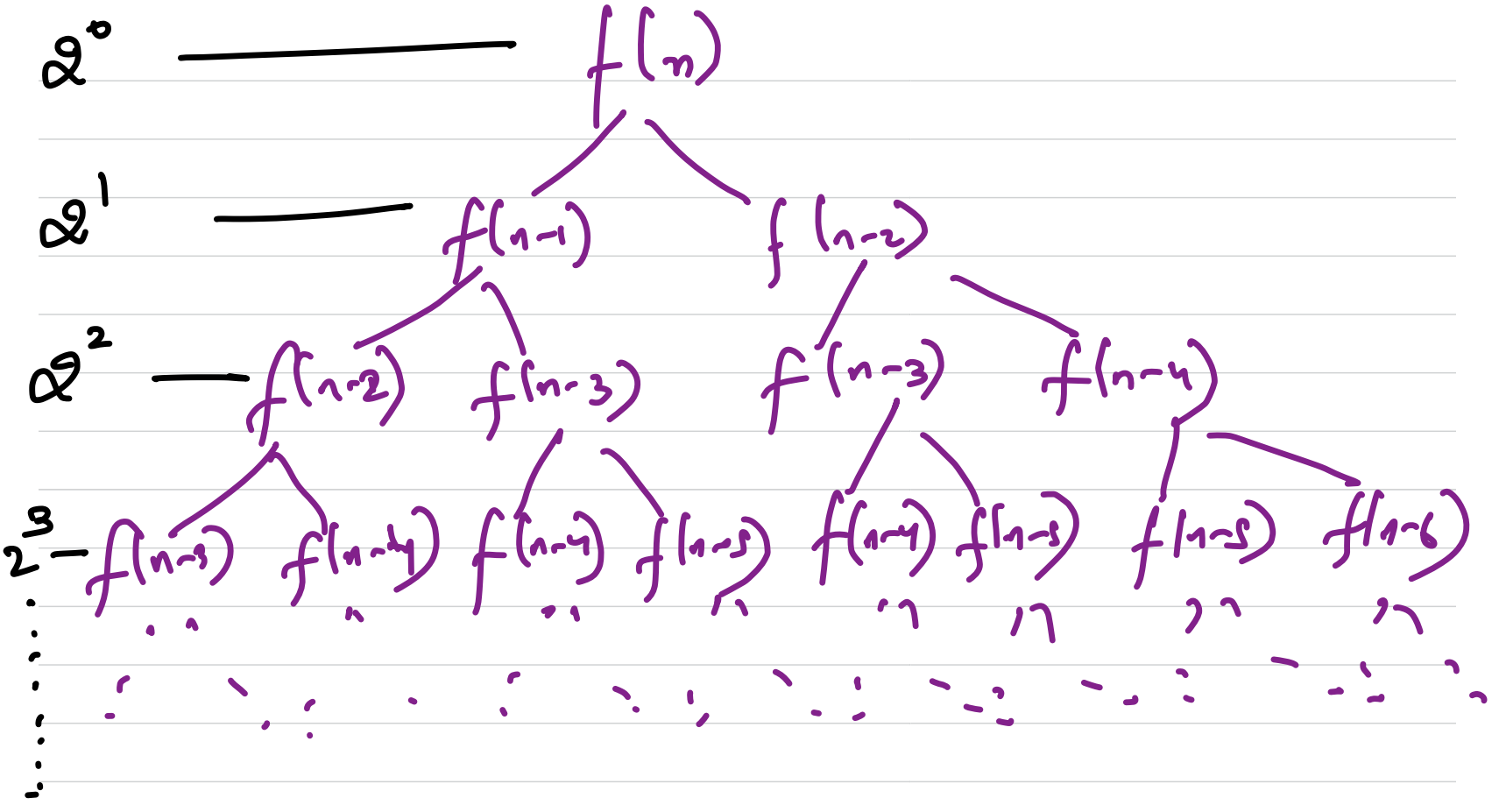
How many function calls.

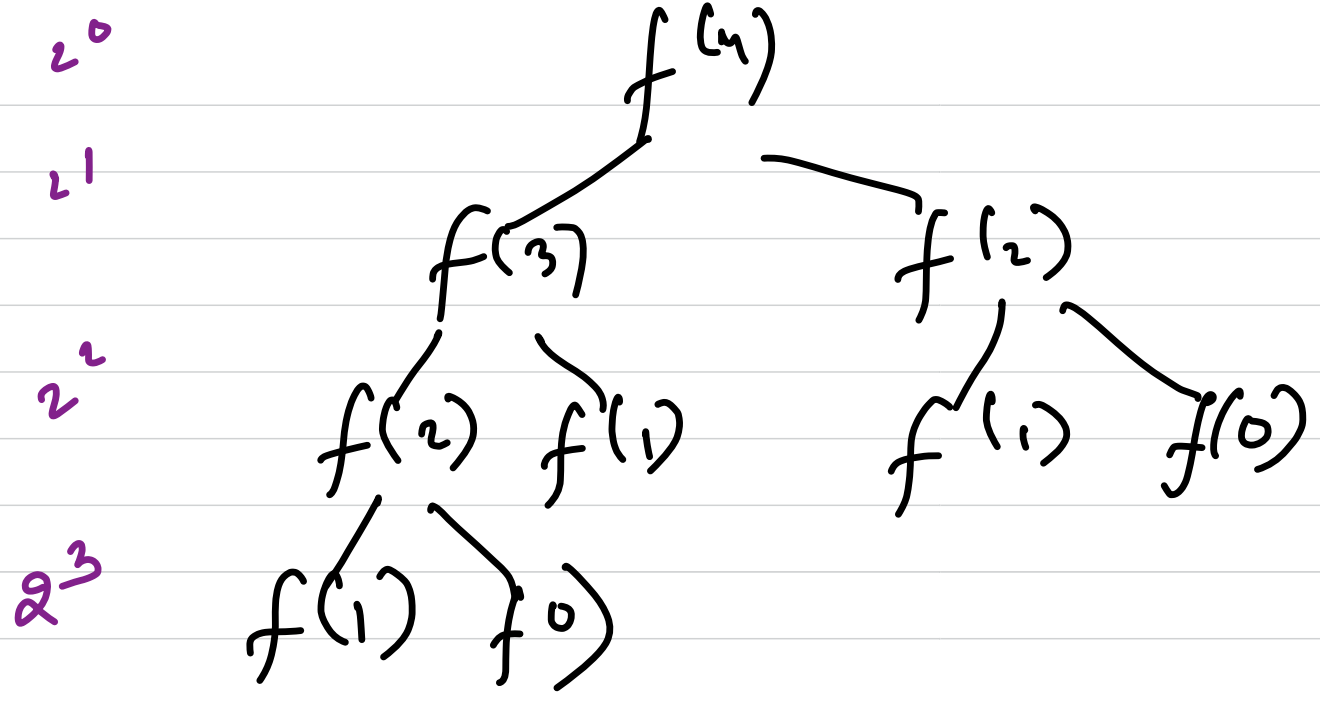
↳ $2^0 + 2^1 + 2^2 \cdots\cdots\cdots 2^n$

→ $S = \frac{2^0 \times (2^{n+1} - 1)}{2 - 1}$

$$S \rightarrow 2^{n+1} - 1$$

time → $O(1) \times (2^{n+1} - 1) \Rightarrow O(2^n)$


2^0
$f(n)$
2^1
$f(n-1)$
$f(n-2)$
2^2
$f(n-2)$
$f(n-3)$
$f(n-3)$
$f(n-4)$
2^3
$f(n-3)$
$f(n-4)$
$f(n-4)$
$f(n-5)$
$f(n-4)$
$f(n-5)$
$f(n-5)$
$f(n-6)$


2^0
2^1
2^2
2^3
f(4)
f(3)
f(2)
f(2)
f(1)
f(1)
f(0)
f(1)
f(0)

1 func call → 1 + 1 + nk + 1 + 1 → $O(nk)$

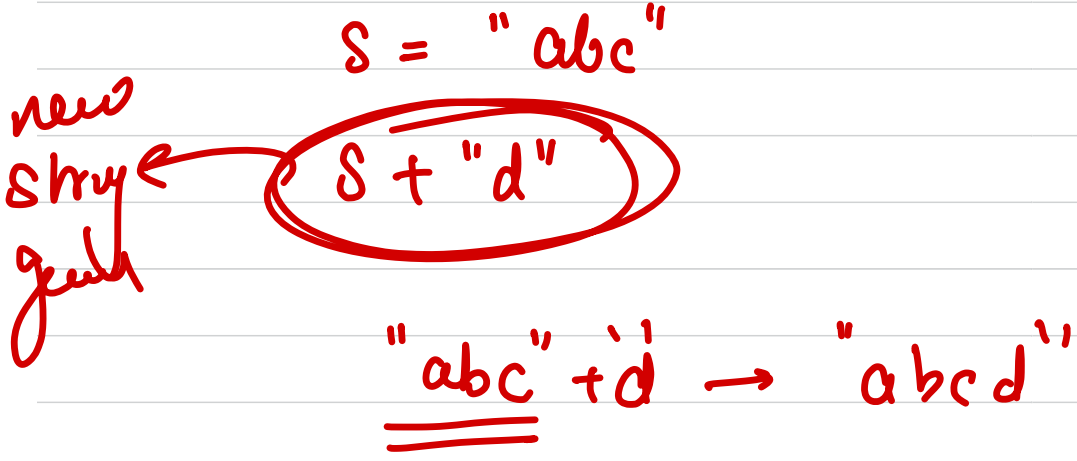
How many times the func is called ?? → n

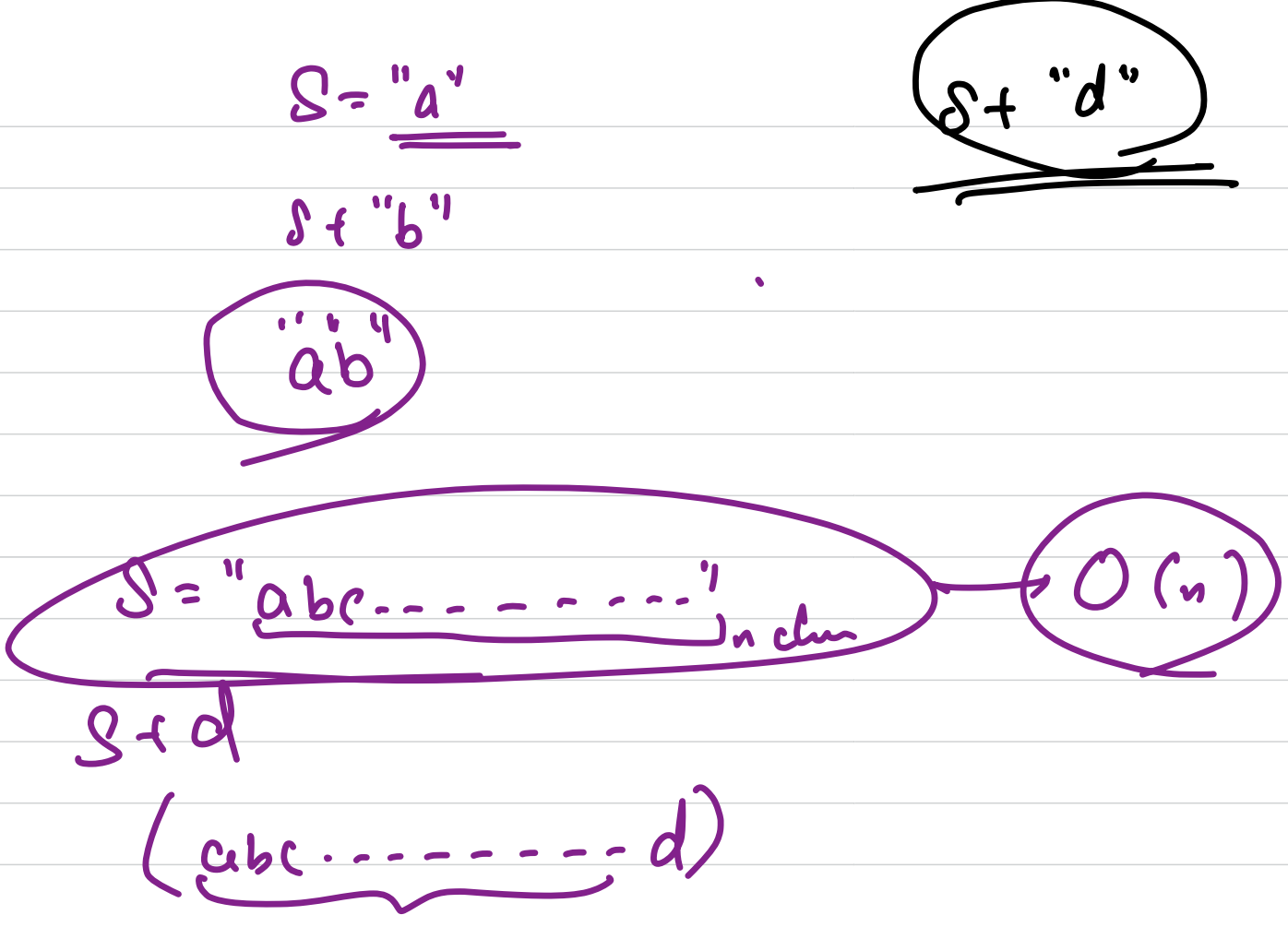
$f(n) \rightarrow f(n-1) \rightarrow f(n-2) \; - - - - - f(0)$

Time → n x $O(nk)$

→ $O(n^2k)$ → $O(n^3)$

In JS strings are immutable




S = "a"
S + "b"
"ab"
S + "d"
S = "abc.....n chu
O(n)
S + d
(abc.....d)


```
S += "def"
```

```
S = S + "def"
```

```
S = "...def"
```

#Dice Combinations

$f(n) \Rightarrow f(n-1) + f(n-2) + f(n-3) + f(n-4) + f(n-5) + f(n-6)$

return the no. of ways to divide n into dice nos

if(n==1) ⟶ return 1;

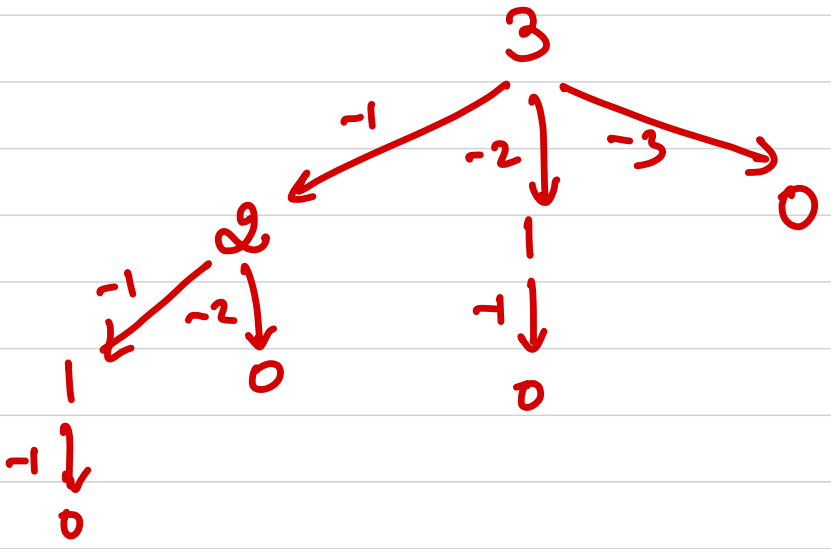
X Better base Case

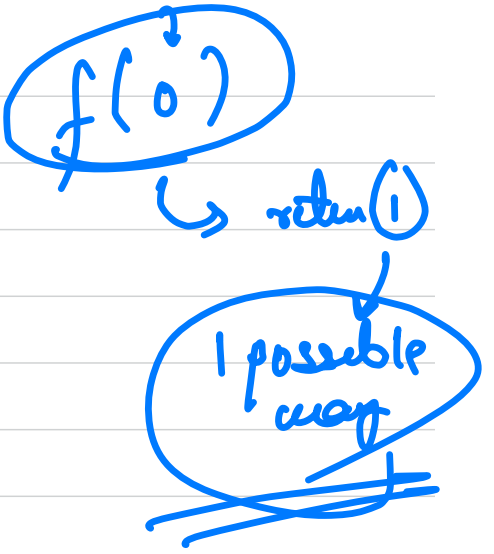
if(n==0) ⟶ return 1;

n ⟶ 0

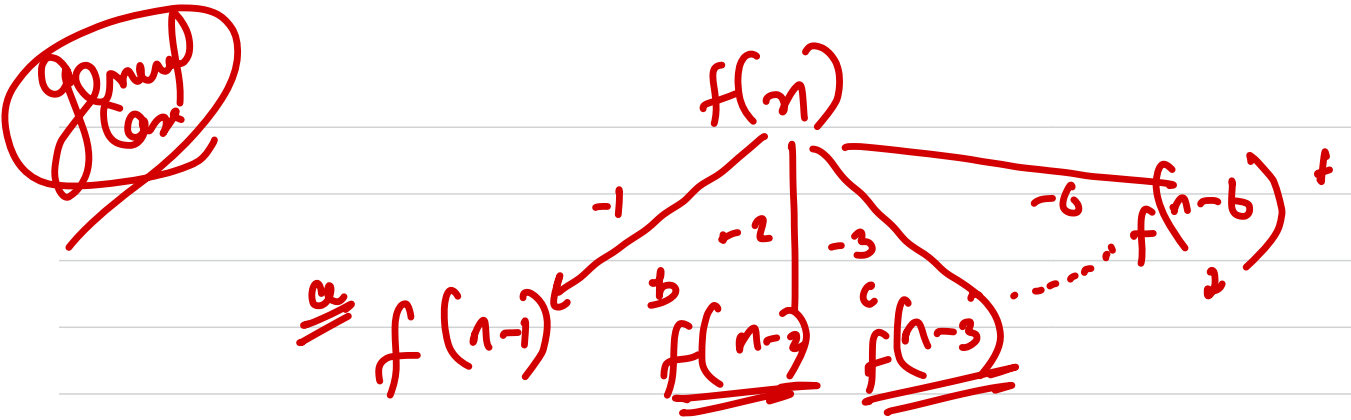
f(n) → f(0)

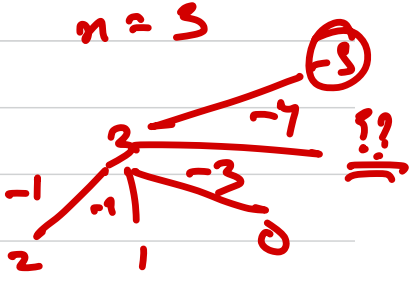
1, 2, 3, 4, 5, 6



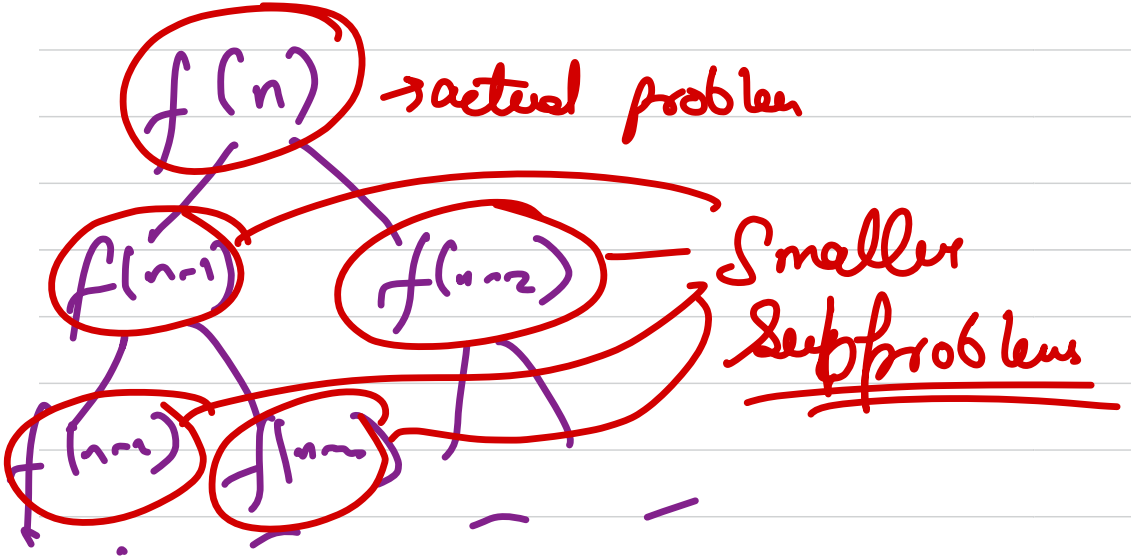


Whenever we reach zero by reducing n, we say we found one possible way.


f(n)
-1
-2
-3
-6
f(n-1)
f(n-2)
f(n-3)
f(n-6)
a
b
c
d
f


n = 5
-3
-7
!!
-1
-1
-3
2
1
0

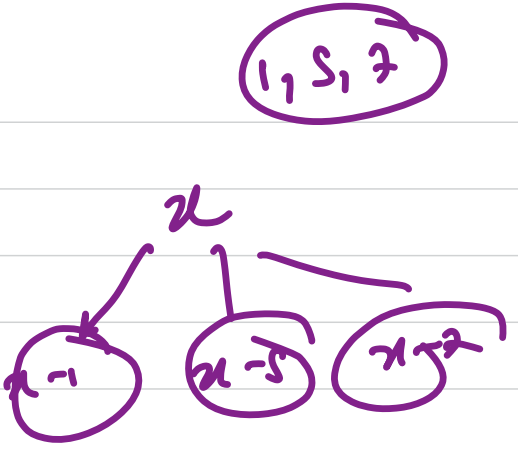
★ State of a problem → It is a subproblem only



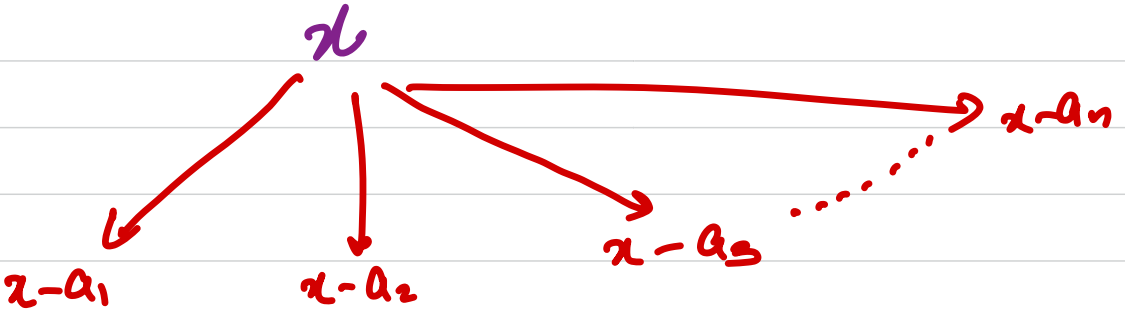
coins array

$f(x, \text{coins})$

returns min no. of coins to make a value x



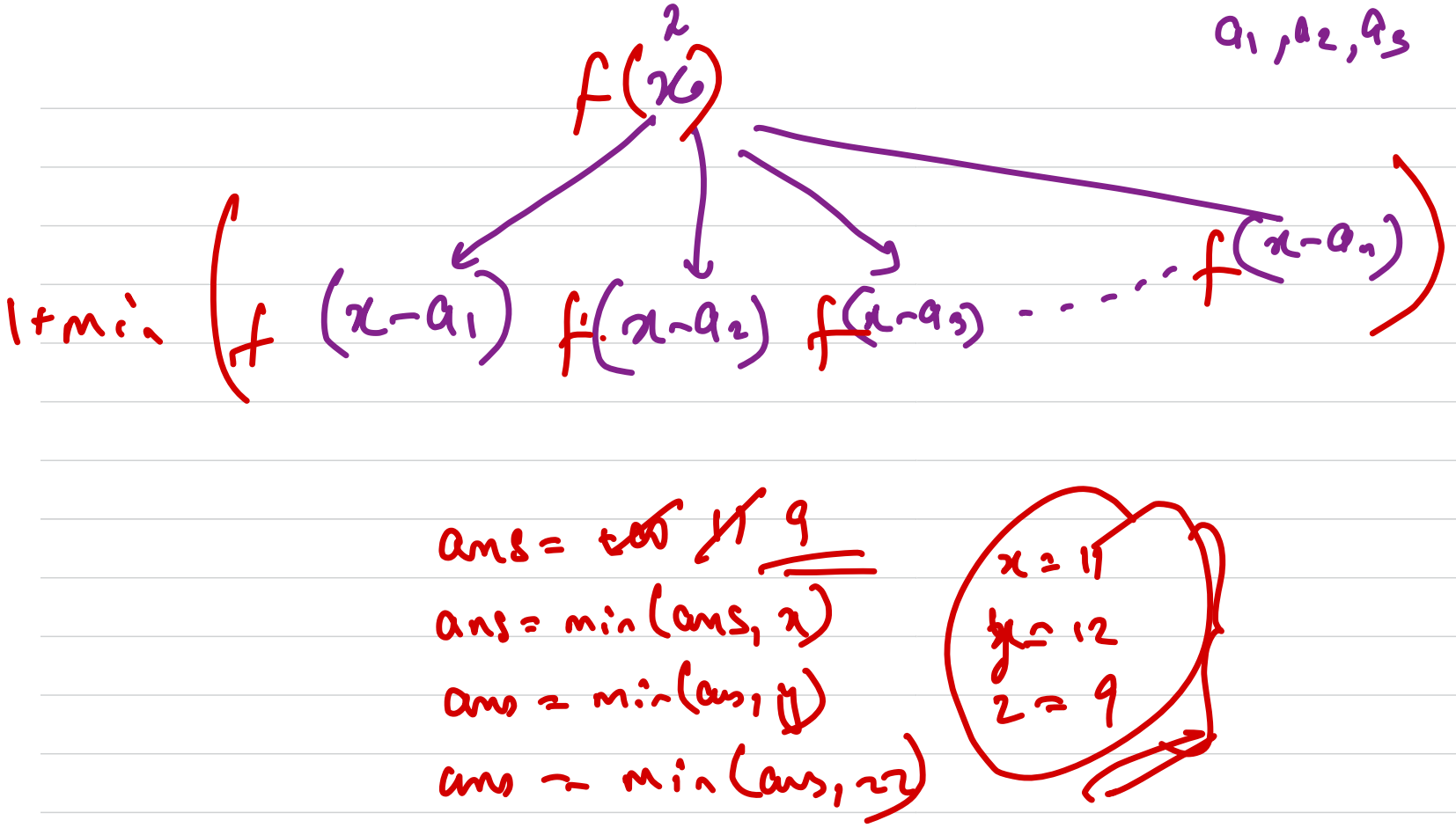
coins → $[a_1, a_2, a_3 \ldots a_n]$

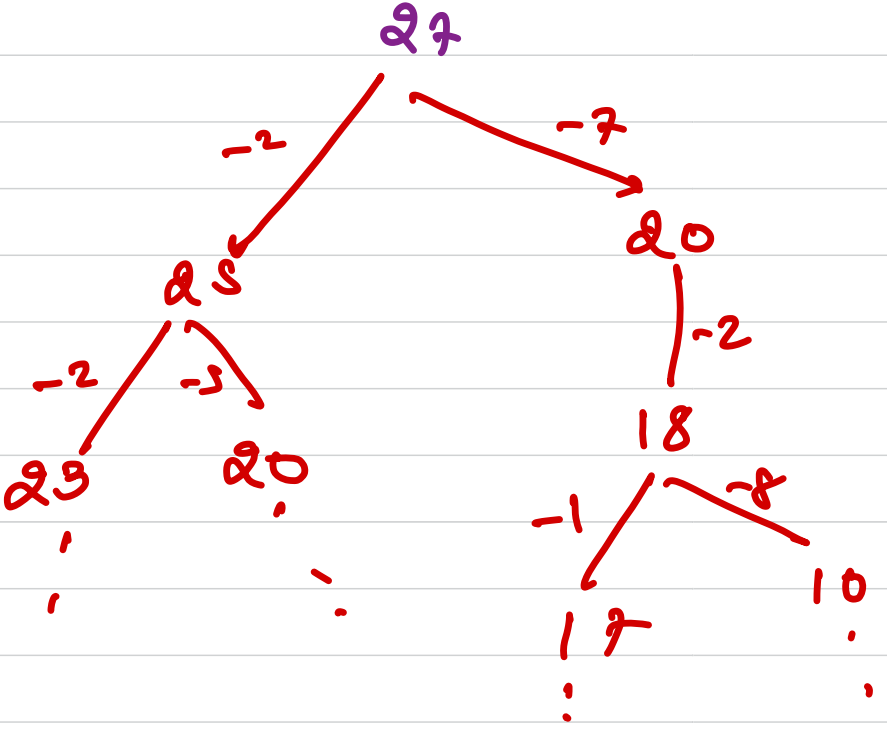


$f(x, \text{coins}) = 1 + \min\left(f(x - \text{coins}[0]), f(x - \text{coins}[1]), f(x - \text{coins}[2]) \ldots\ldots\right)$

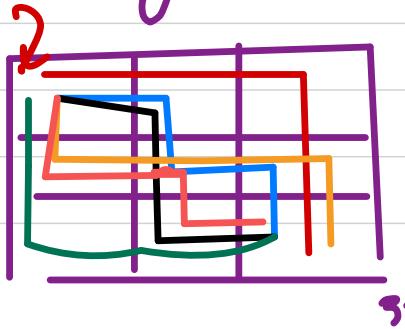
↓

return the min no of coins to make x


a_1, a_2, a_3
$f(x)$
$1 + \min\left(f(x-a_1), f(x-a_2), f(x-a_3), \ldots, f(x-a_n)\right)$
ans = 9
ans = min(ans, x)
ans = min(ans, 11)
ans = min(ans, 12)
x = 11
x = 12
2 = 9


27
-2
-7
25
20
-2
-3
-2
23
20
18
-1
-8
17
10

Q. Given a grid of n x m dimensions - You're standing at 0,0 (i.e. top left corner of grid). From any cell of the grid you can either go right or go downwards. Find the total no. of ways to reach the bottom right corner of the grid from the top left corner.



ans → 6

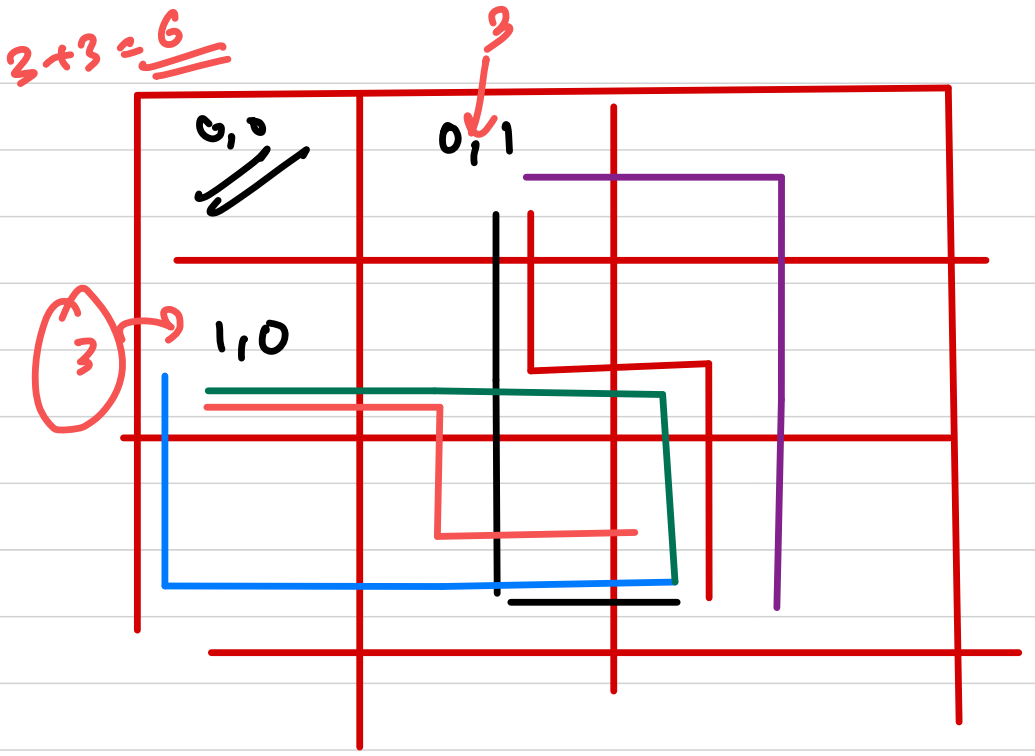
$f(i,j) = f(i,j+1) + f(i+1,j)$

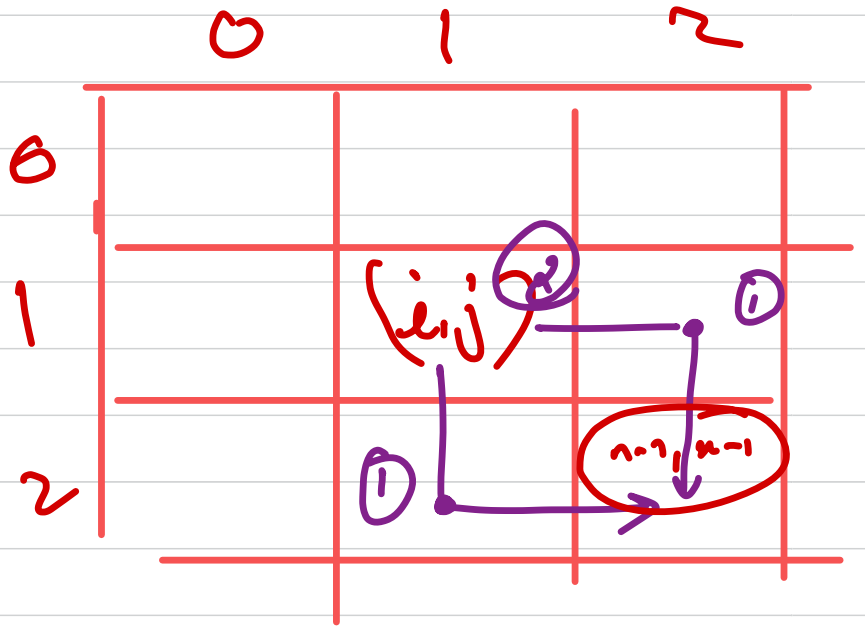
$f(i,j)$ → return the no. of ways to reach bottom right from i,j

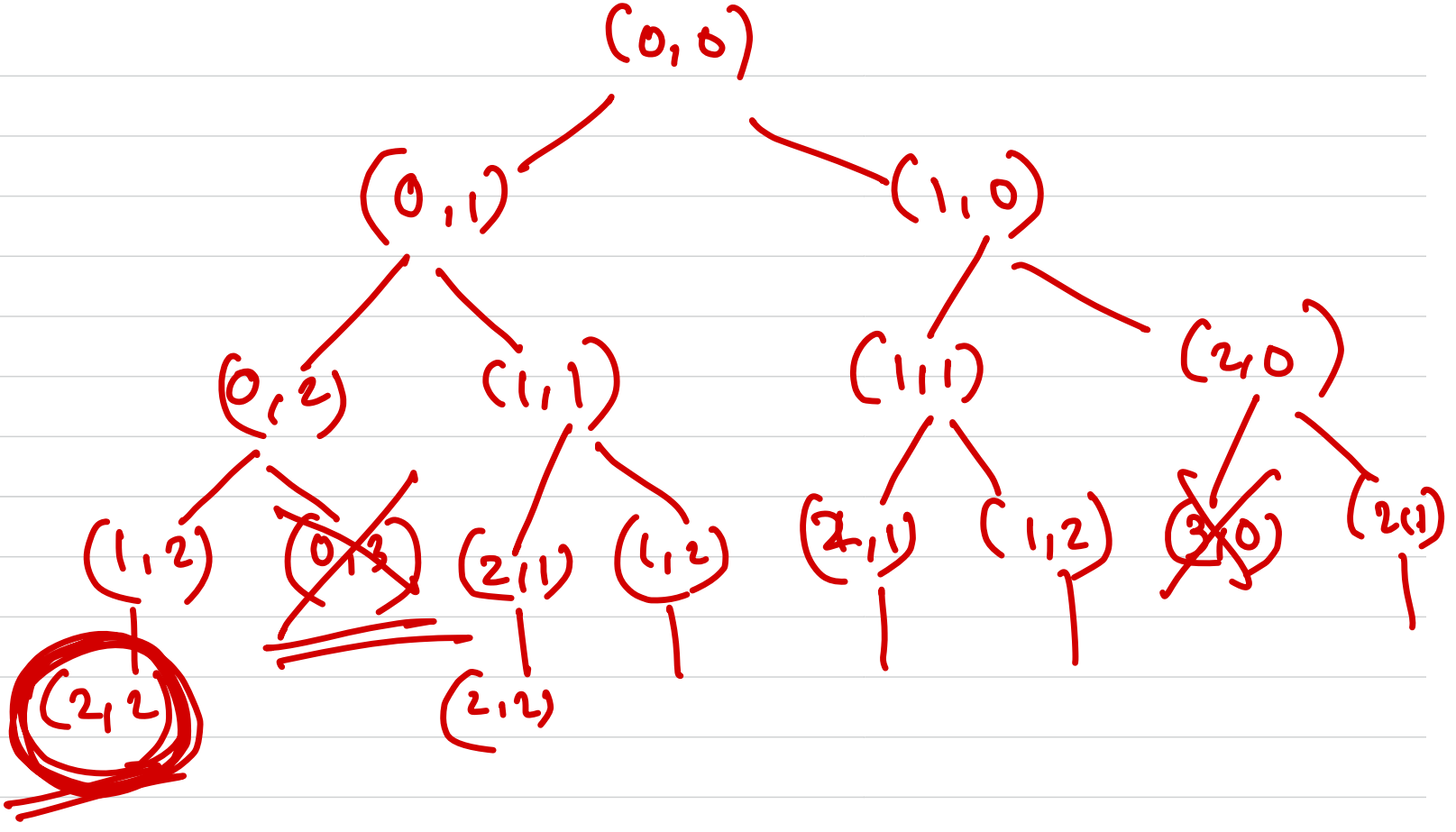
$f(i,j+1)$ → no. of ways to reach bottom right from rightward cell to i,j

$f(i+1,j)$ → no. of ways to reach bottom right from downward cell to i,j

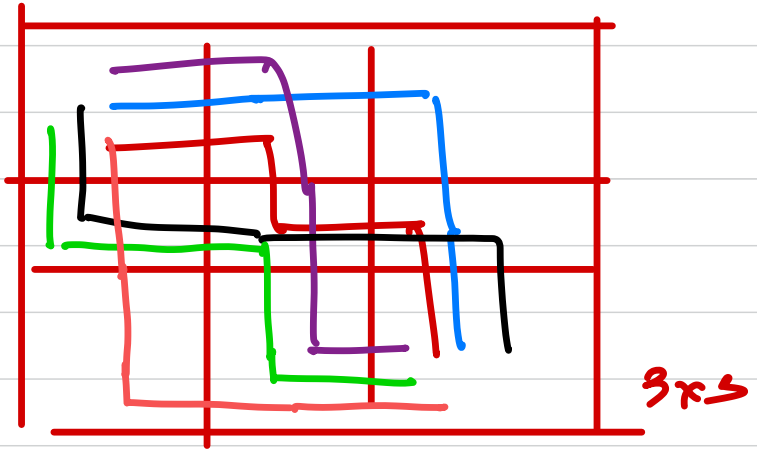
ans → $f(0,0)$


3+3=6
3
0,0
0,1
3
1,0


0
1
2
0
1
2
(i,j)
2
1
1
n-1, n-1


(0,0)
(0,1)
(1,0)
(0,2)
(1,1)
(1,1)
(2,0)
(1,2)
(0,3)
(2,1)
(1,2)
(2,1)
(1,2)
(3,0)
(2,1)
(2,2)
(2,2)

Instead of just count, can we also print the path. ?



R R D D

R D R D

R D D R

D R R D

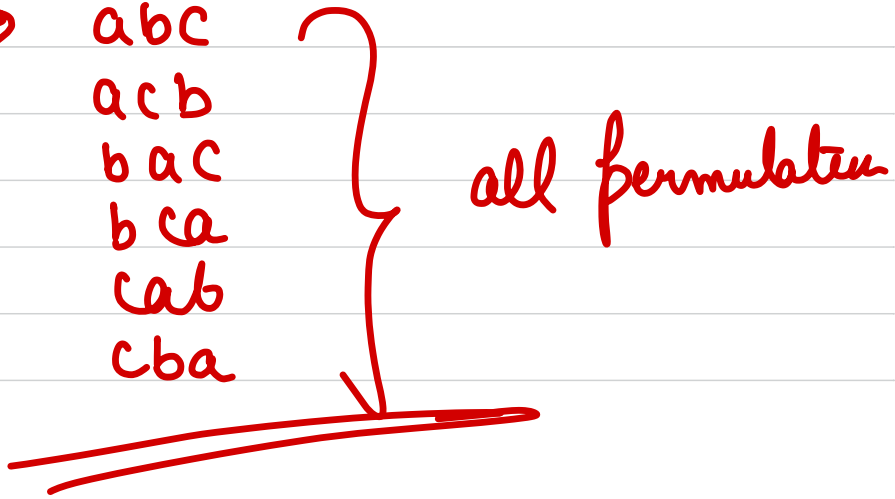
D R D R

D D R R

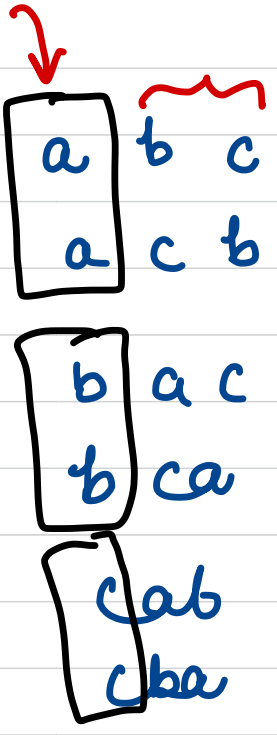
Q» Given an array of characters, print all permutations of the array (recursively)

Ex ['a', 'b', 'c']

↳ abc
acb
bac
bca
cab
cba



↳ [a, b, c]



[a, b, c, d]

a b c d
a b d c
a d b c
a d c b
a c b d
a c d b

b a c d
b a d c
b d a c
b d c a
b c a d
b c d a

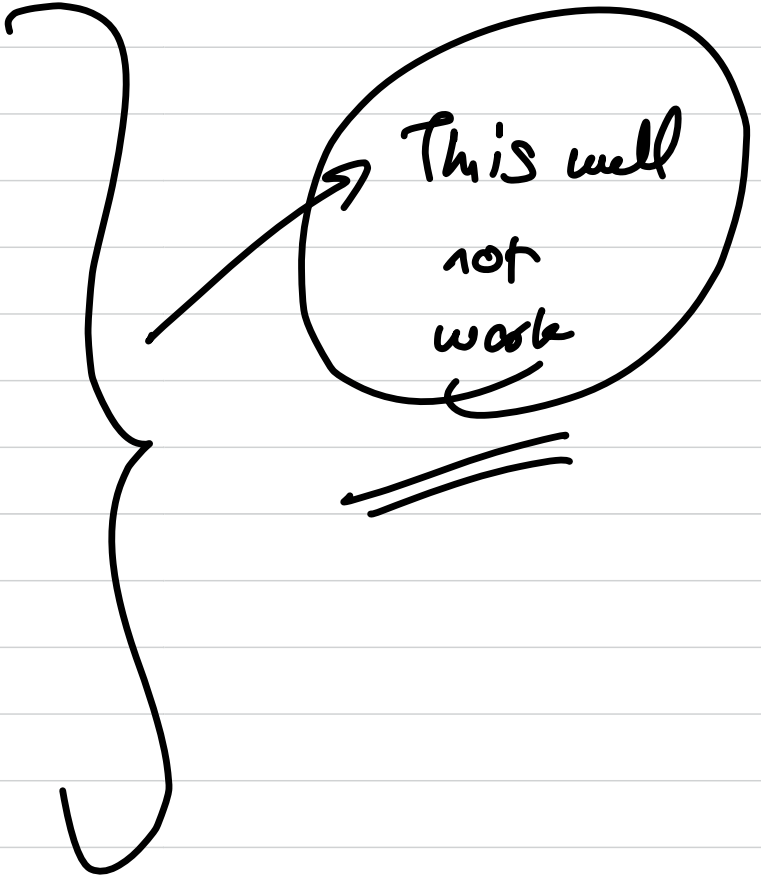
c →

d →

f(arr, idx, output)

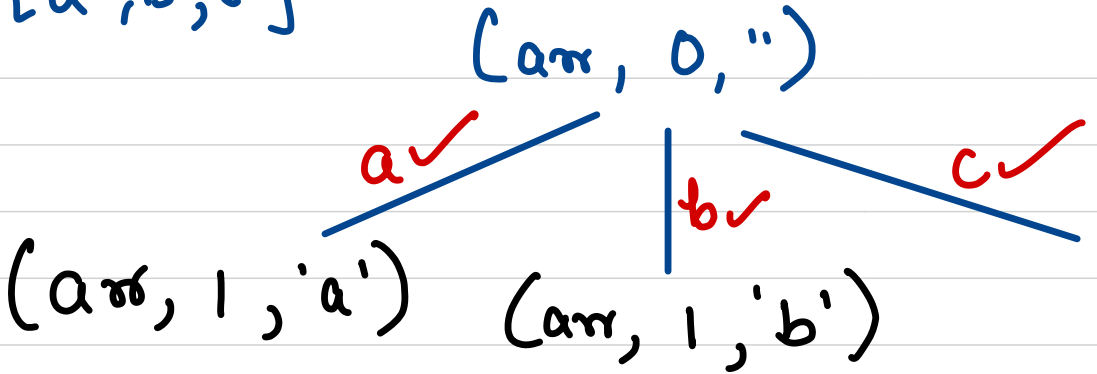
↓

this function returns permutation of the elements starting from index idx.



answer → f(arr, 0, "")

[a, b, c]



[every char is gonna get a chance to become the first char.]

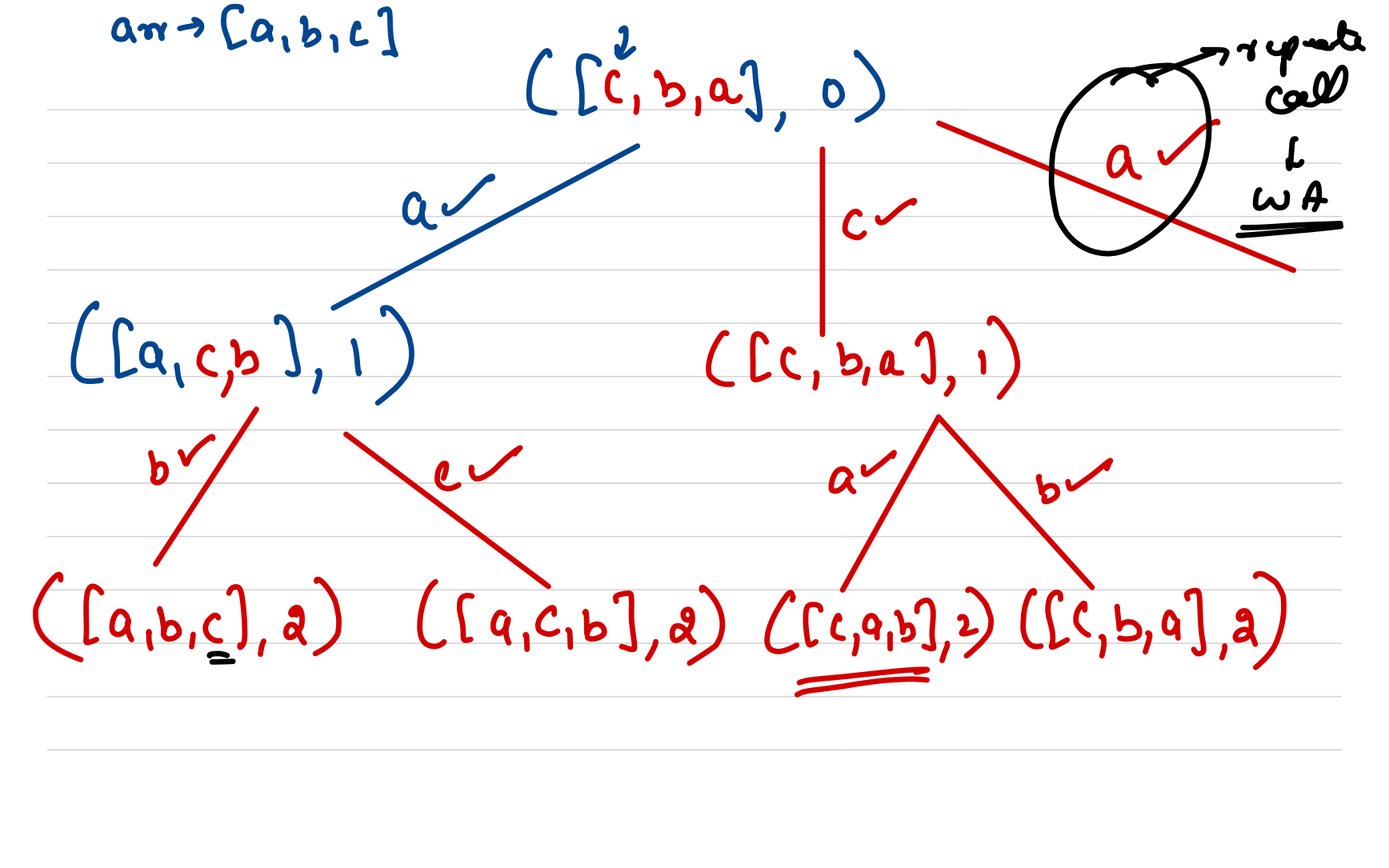
f(arr, idx)

↓

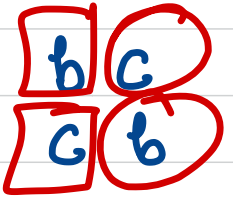
return all permutations

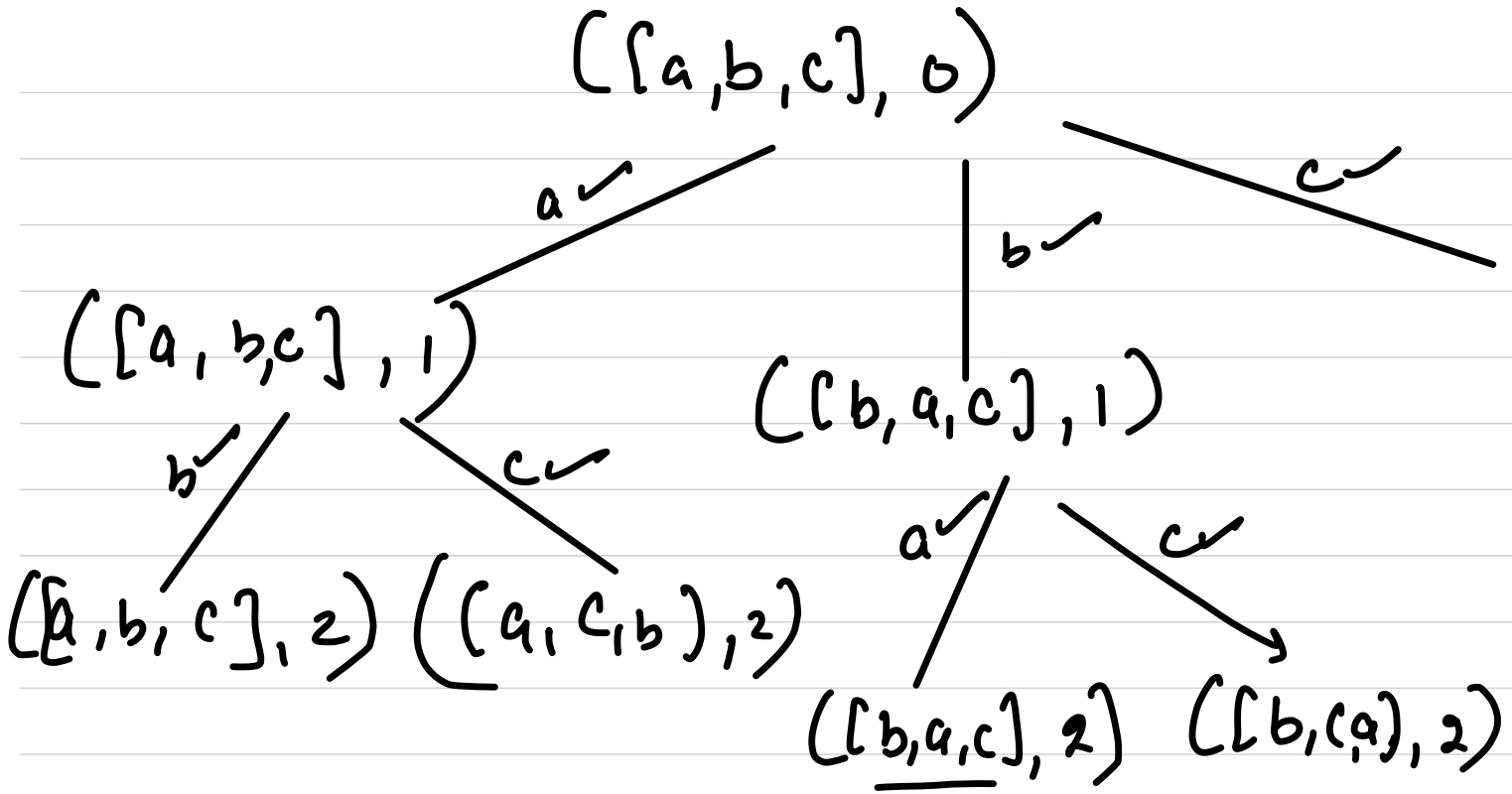
starting from index idx

arr → [a, b, c]


([c, b, a], 0)
a
c
([a, c, b], 1)
([c, b, a], 1)
b
c
a
b
([a, b, c], 2)
([a, c, b], 2)
([c, a, b], 2)
([c, b, a], 2)
a
recursive call
↓
WA

bc →





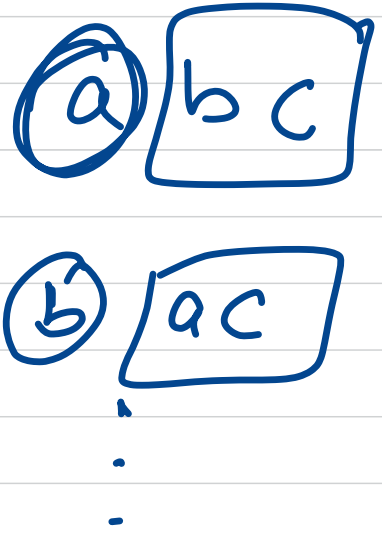
abc, acb, bac, bca

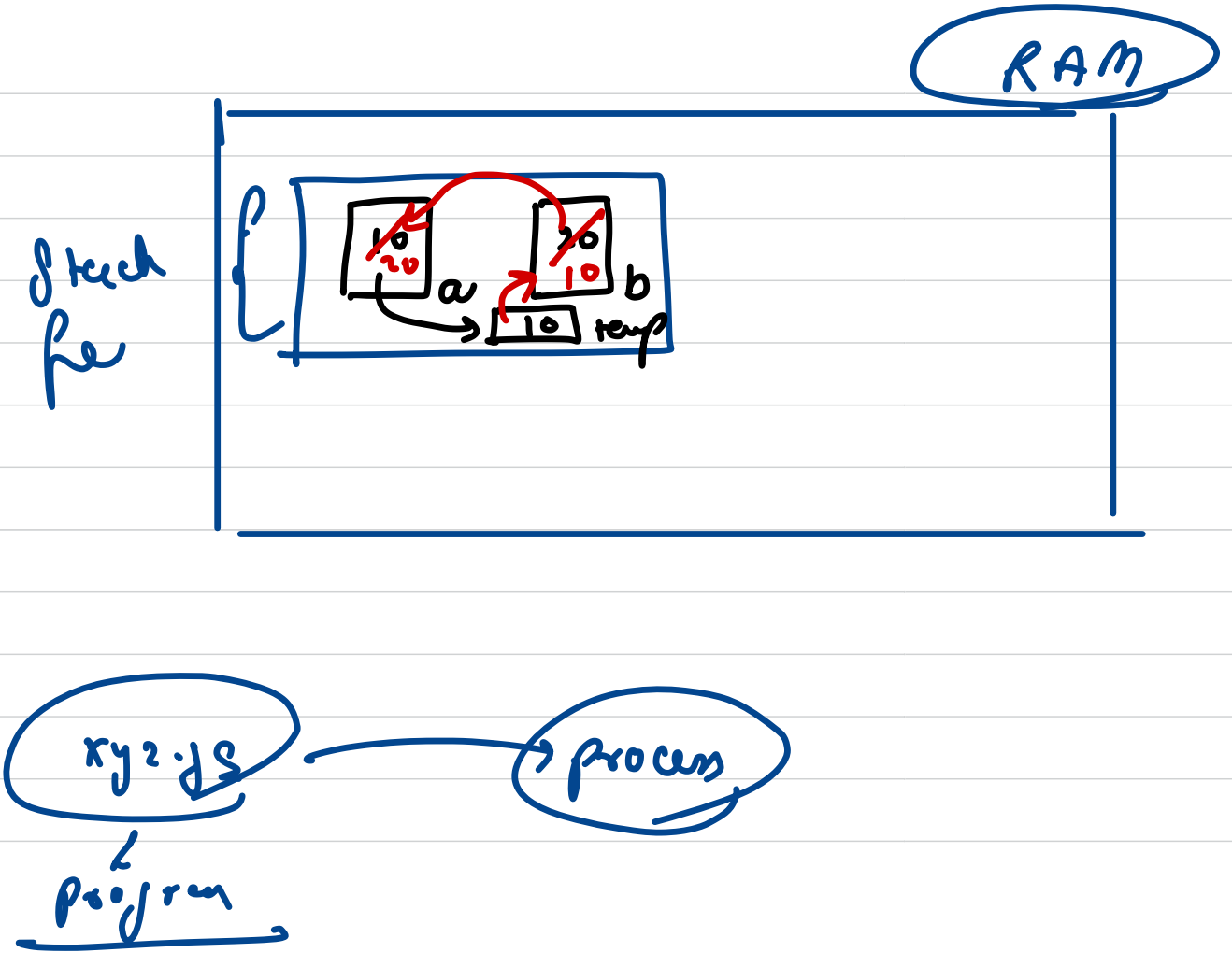
swap( )

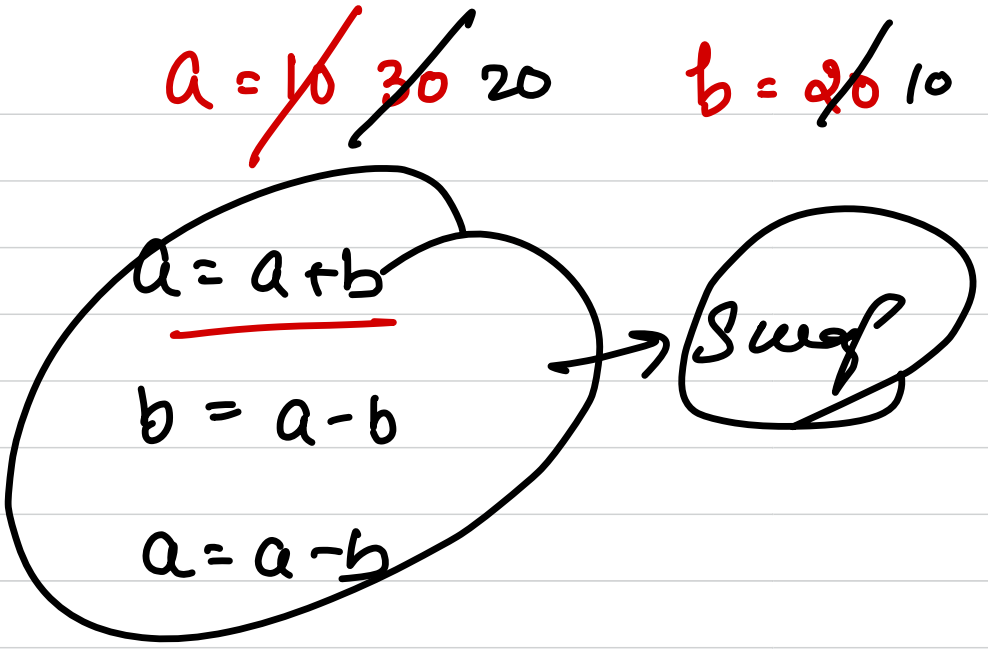
f( )

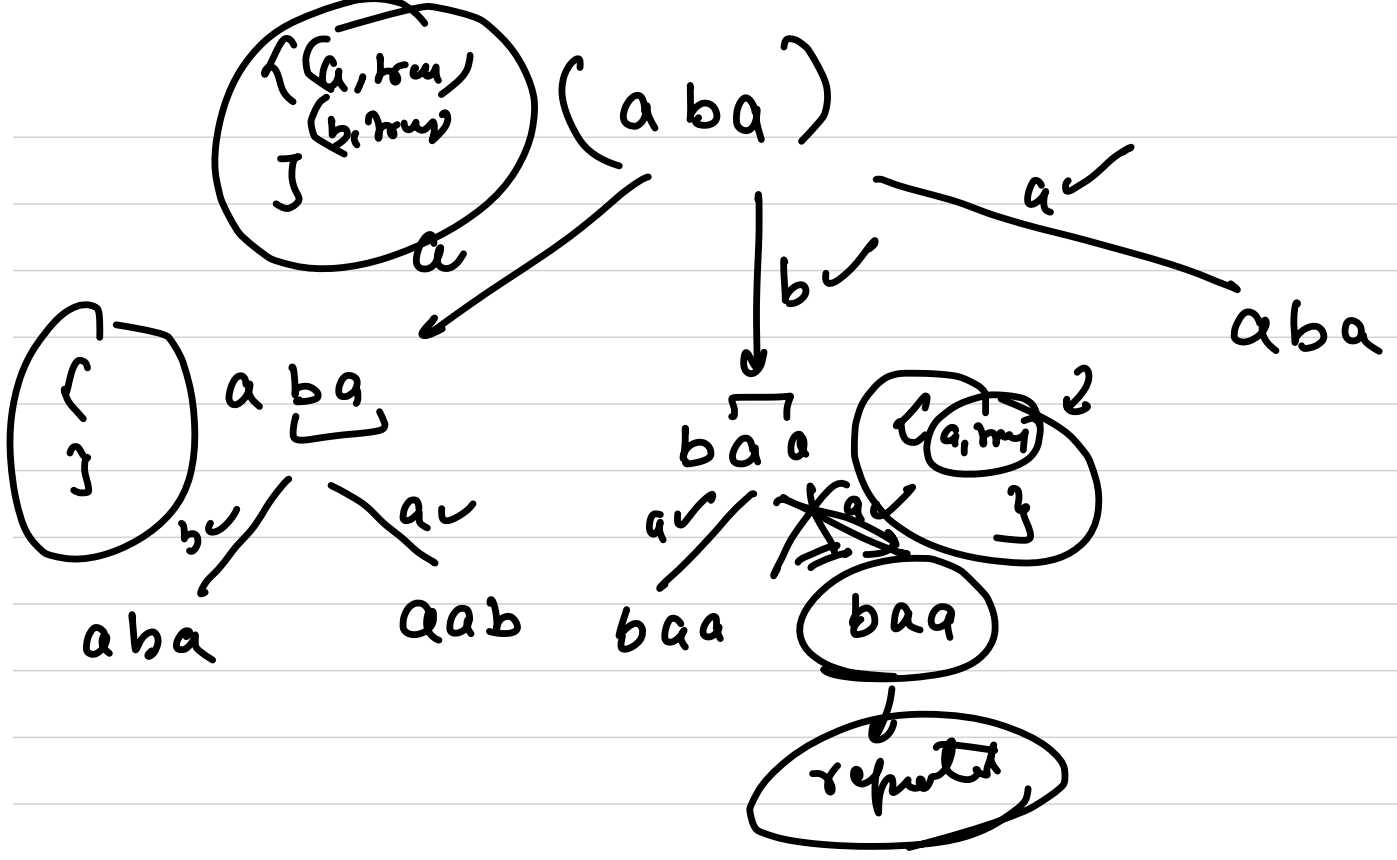
swap( )

#Self work → To give every char a chance to become the first char.




RAM
Stack
fn
10
20
a
20
10
b
10
temp
xyz.js
process
program

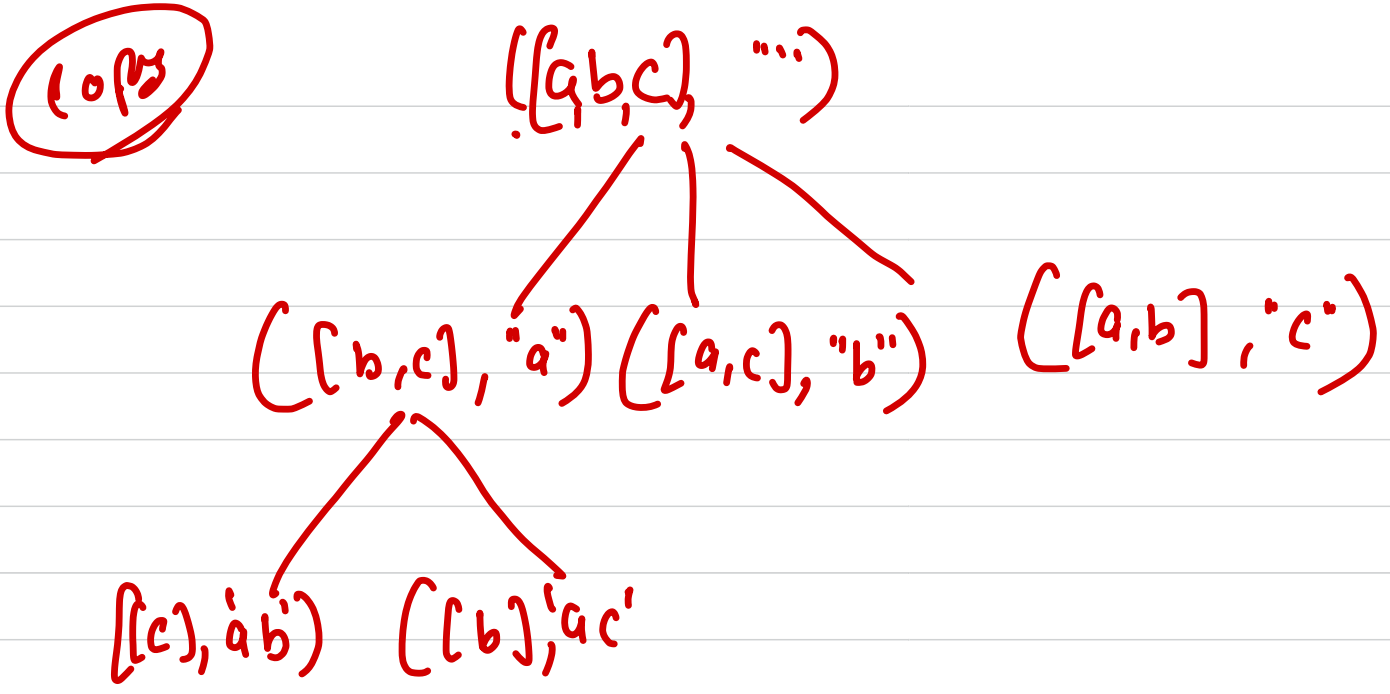

a = 10 30 20
b = 20 10
a = a + b
b = a - b
a = a - b
Swap


{(a, true)
(b, true)
}
aba
a
b
a
{
}
aba
b
a
aba
aab
baa
a
a
baa
{(a, true)}
2
baa
repeat

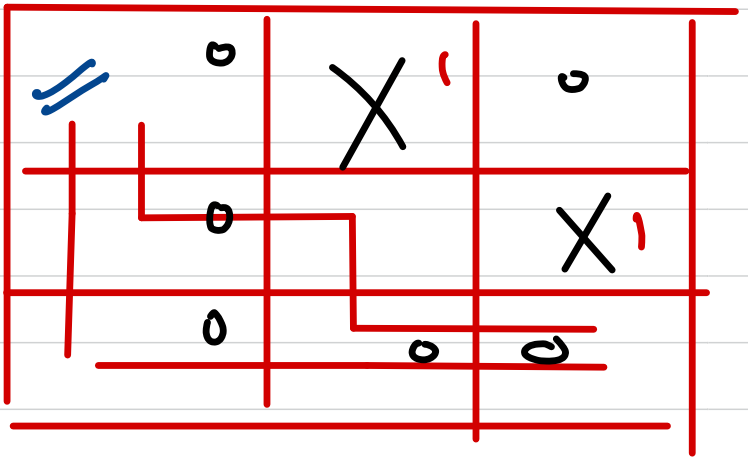
= {

key - value

}


([a,b,c], "")
([b,c], "a")
([a,c], "b")
([a,b], "c")
([c], "ab")
([b], "ac")

Q: Given a matrix of m x n. You're starting from top left. From every cell you can either go right or down. Few cells are open and few are blocked. From blocked cell we cannot go anywhere. All the movement can happen on the open cell. Calc the no. of ways to reach bottom right.

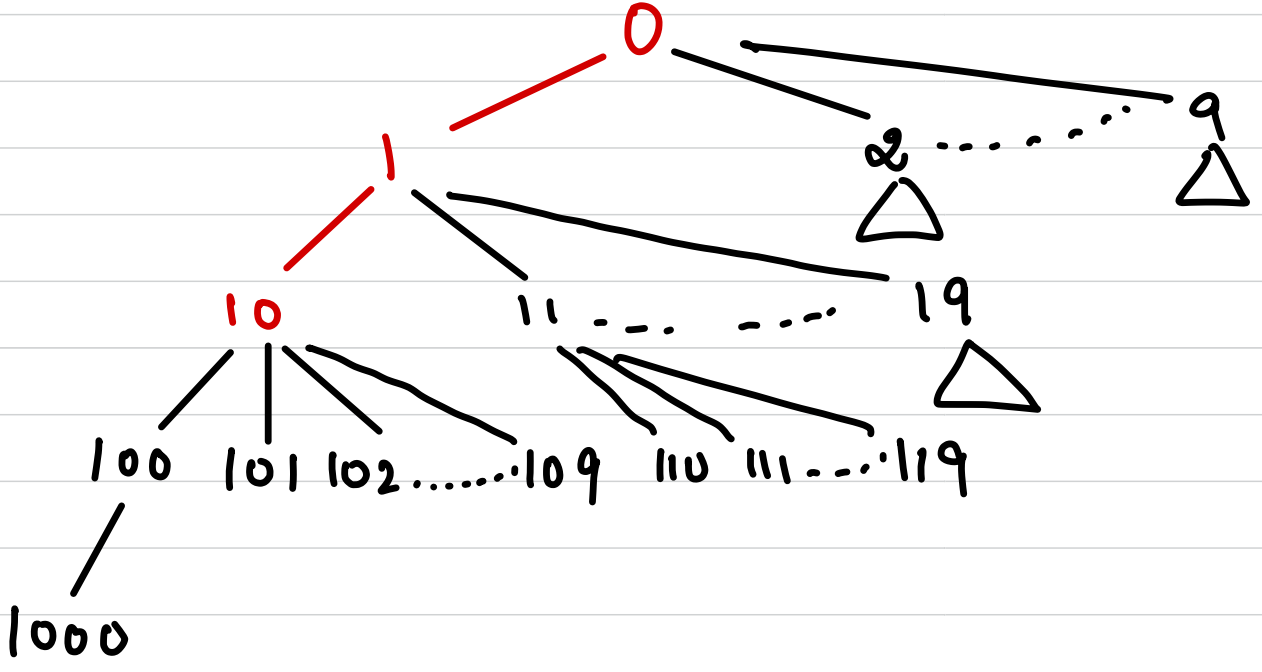


f(i, j, grid)

```
if (grid[i][j] == 1)
    return 0;
```

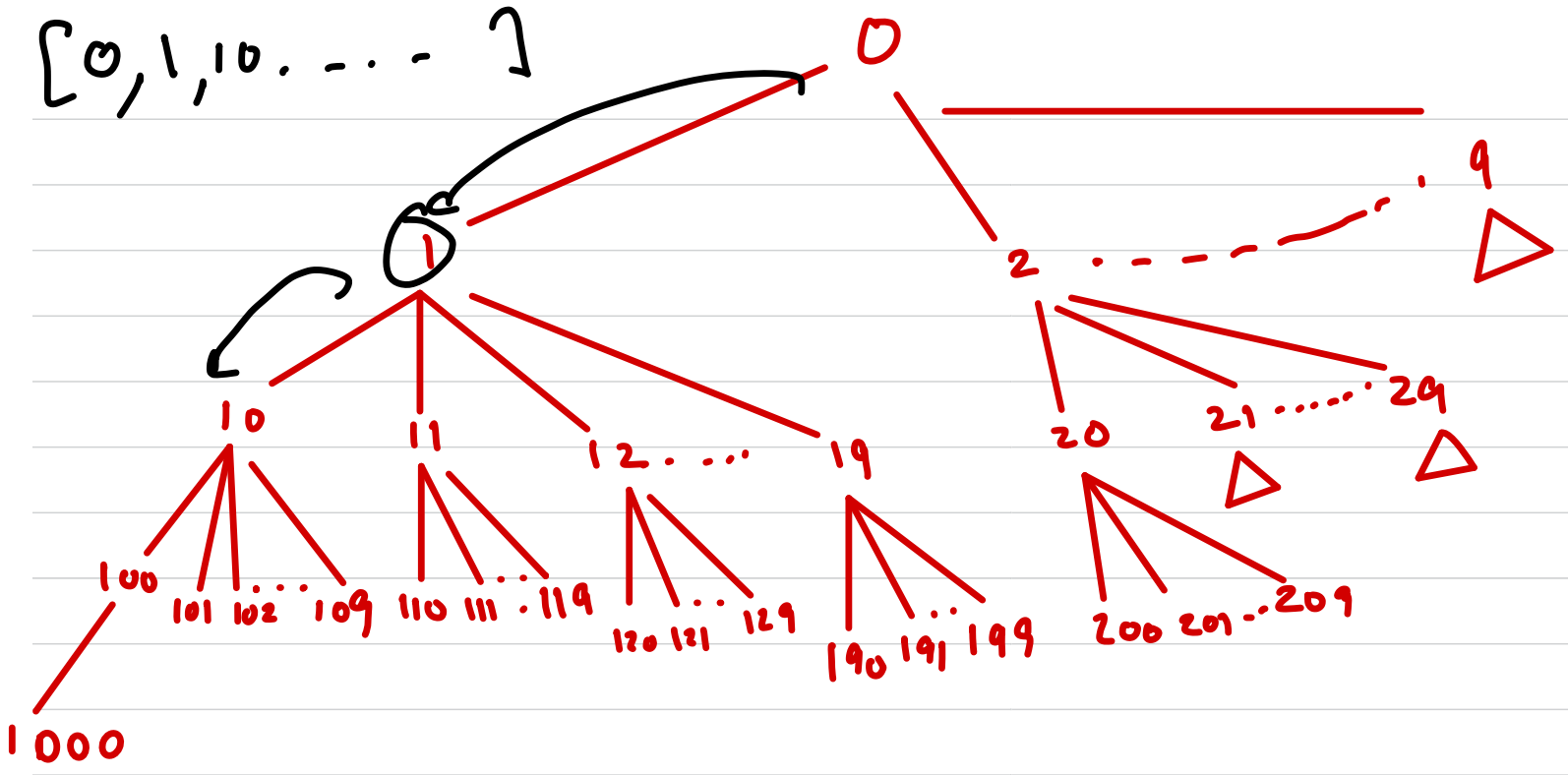
lexicographical order

0 - 1000



0 - a
1 - b
2 - c
3 - d
⋮ ⋮

baaa
bab
bac


[0,1,10...]
0
1
2
9
10
11
12...
19
20
21.....29
100
101
102
109
110
111
119
120
121
129
190
191
199
200
201
209
1000

$f(i, n)$ =

$f(10 \times i + 0, n)$

$f(10 \times i + 1, n)$

$f(10 \times i + 2, n)$

⋮

$f(10 \times i + 9, n)$

prints numbers from i, n in lexico order

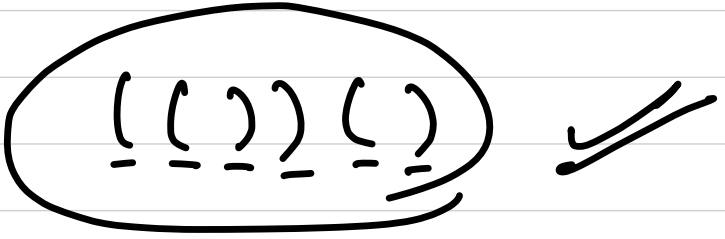
Ans → $f(0, n)$

)(()) ✗

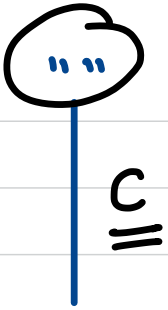
(()) ✓

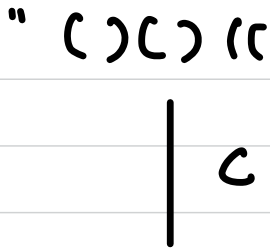
)(✗

)(✗



print all well formed balanced parenthesis of length 2n.





↳ for an empty string or an already balanced string, we can't add closing parenthesis at the end.

if open-parenthesis-count == n, then we can only add closing.

f(open, closing, res, n) =

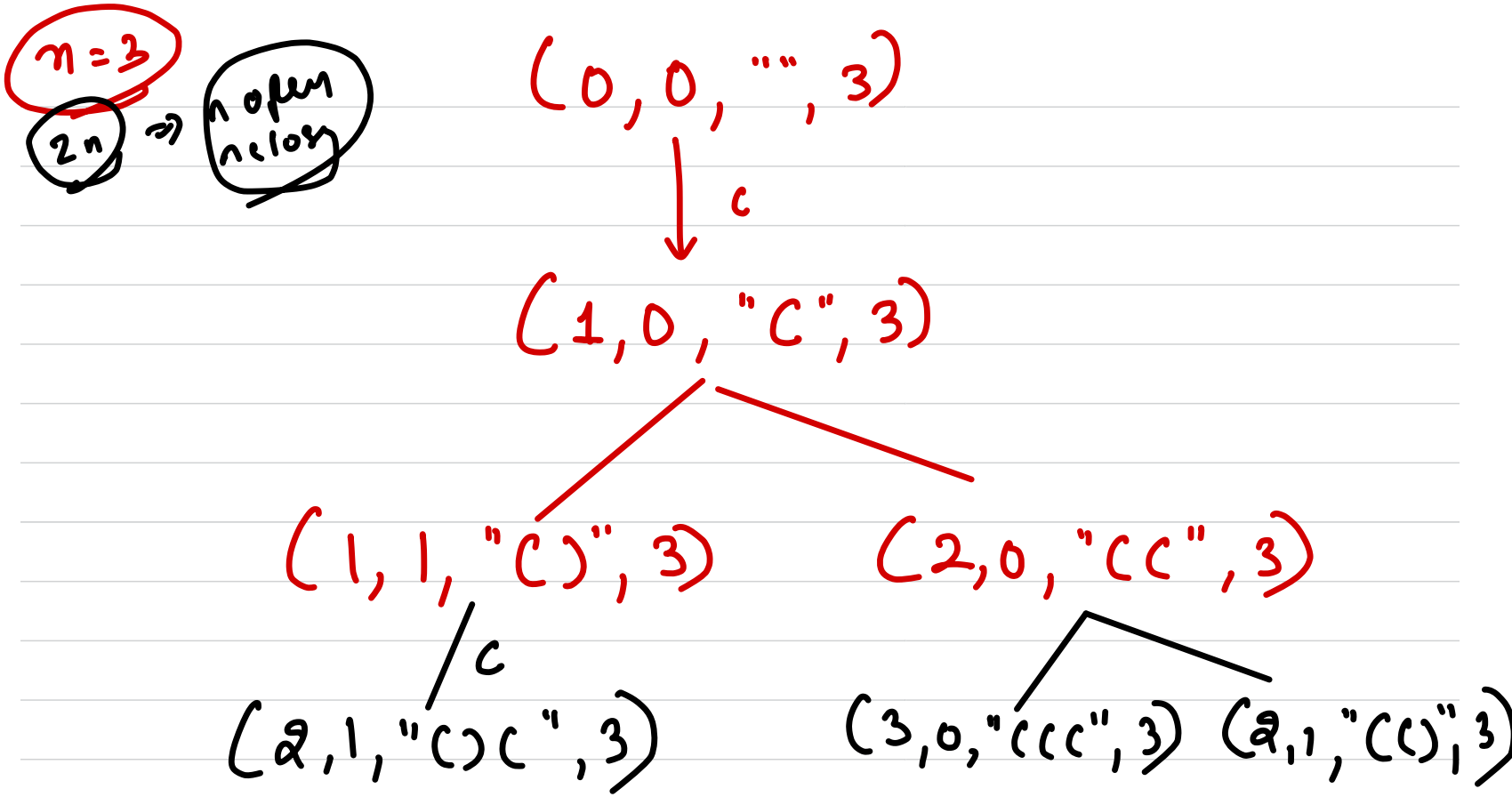
count

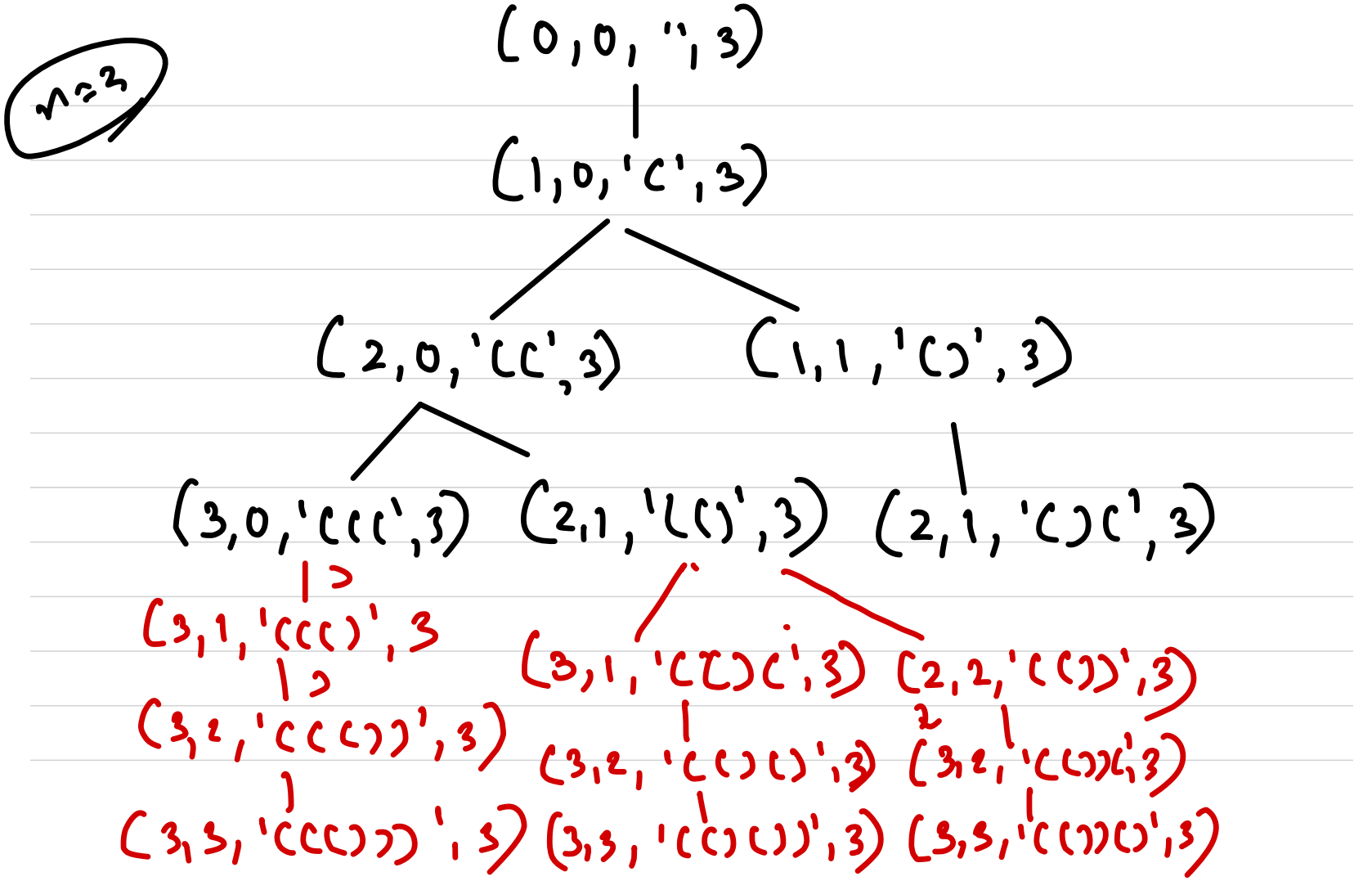
prints balanced strings of length 2n

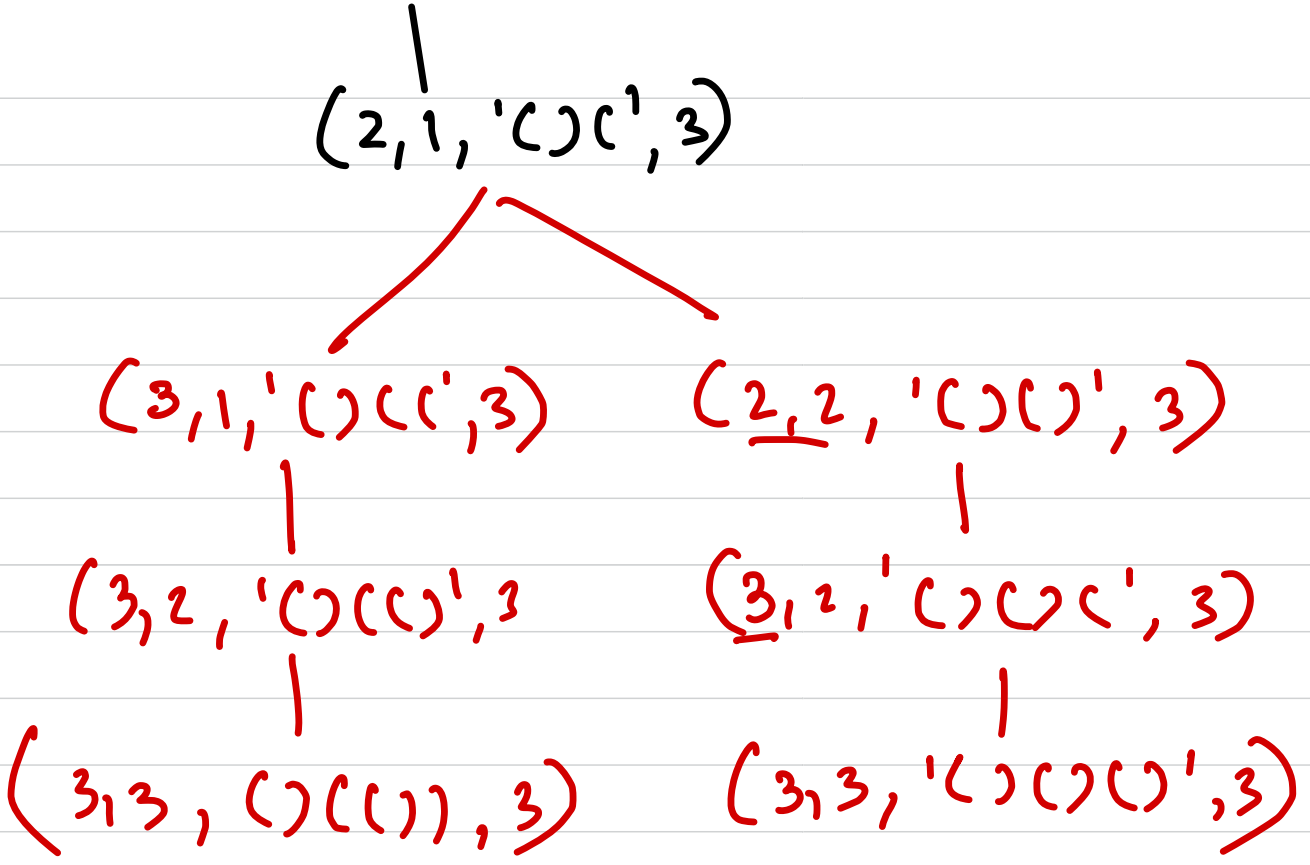
- if open == close and open < n: f(open+1, close, res+"(", n)
- else if open == close: f(open, close+1, res+")", n)
- else: f(open+1, close, res+"(", n) and f(open, close+1, res+")", n)

Base Case

```
if (close == n)
    console.log(res);
```


n = 3
2n
⇒
n open
n close
(0, 0, "", 3)
c
(1, 0, "(", 3)
(1, 1, "()", 3)
(2, 0, "((", 3)
c
(2, 1, "()(", 3)
(3, 0, "(((", 3)
(2, 1, "(()", 3)

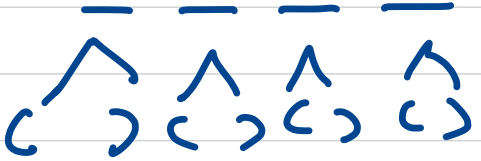

n=3
(0,0,'',3)
(1,0,'(',3)
(2,0,'((',3)
(1,1,'()',3)
(3,0,'(((',3)
(2,1,'(()',3)
(2,1,'()(',3)
(3,1,'((()',3
(3,2,'(((())',3
(3,3,'((()))',3
(3,1,'(()(',3)
(2,2,'(())',3)
(3,2,'(()()',3)
(3,3,'(()())',3)
(3,2,'(())(',3)
(3,3,'(())()',3)


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

So logic is based on the fact that how many opening & closing parentheses you have.

$n = 2$

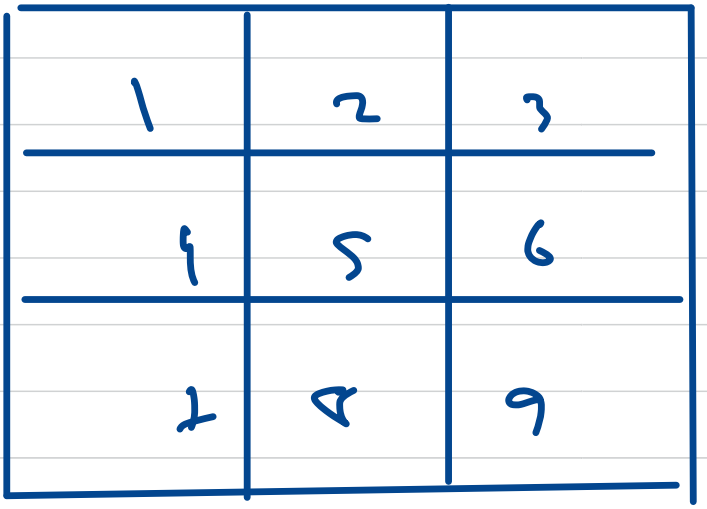
$2n$



$m \leq k \leq n$

k

get me all pattern of length k that starts with 1



all pattern starts with 1 → x

all pattern starts with 2 → y

all pattern starts with 5 → z

ans → $4 \times x + 4 \times y + z$

m = 2

n = 10

[2, 10]

$m \le k \le n$

k = 2,
3,
4,
5,
6,
7,
8,
9,
10

add

for (k=m; k<=n; k++)

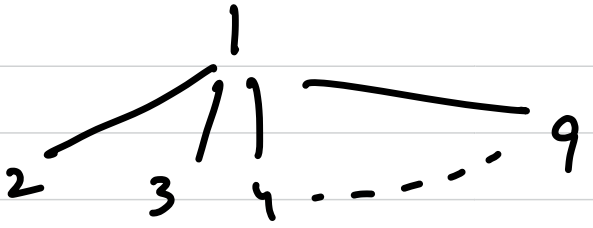
→ k

$f(k)$

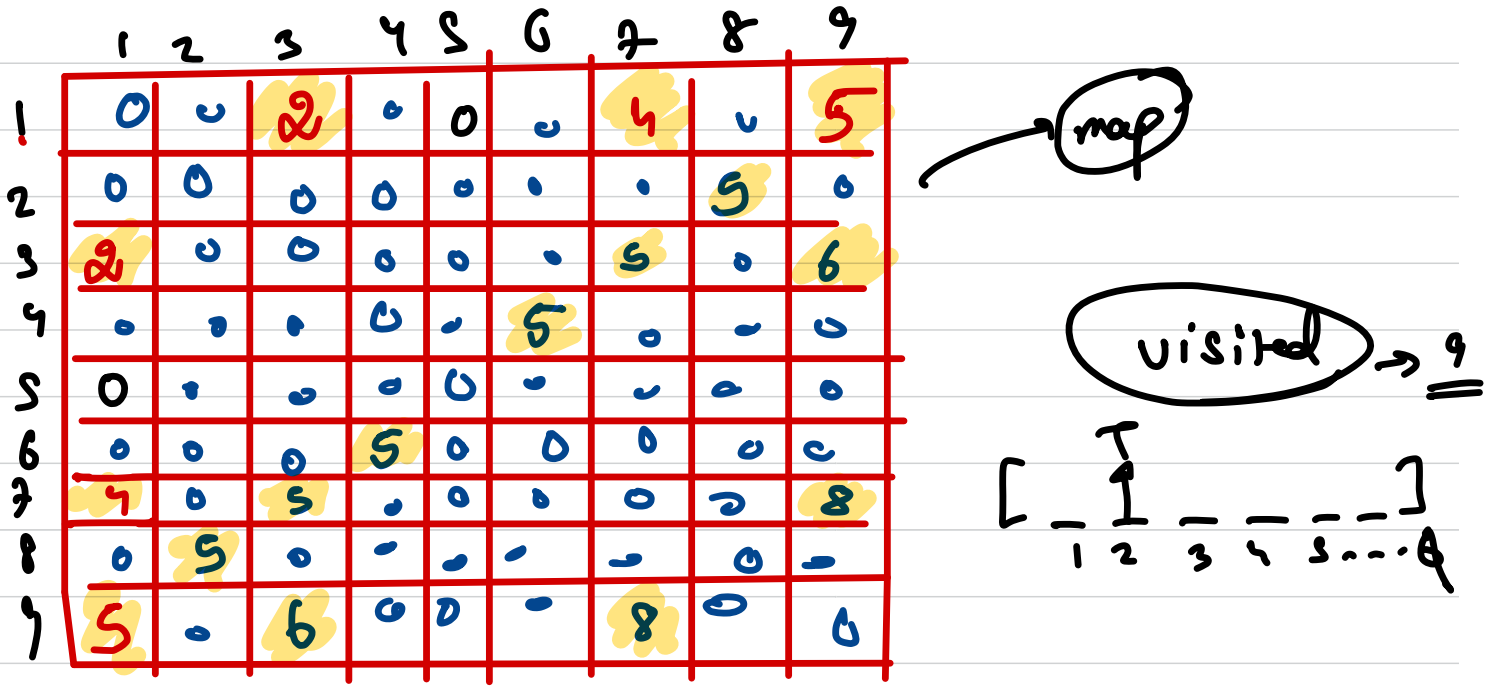
↳ all the unlock pattern of length k

$$\sum_{k=m}^{n} f(k)$$

```
for (k=m; k<=n; k++)
    ans += f(k)
```



mapping


map
visited
9
1 2 3 4 5 ... 9

$$f(\text{visited}, \text{num}, k) = \sum f(\text{visited}, j, k-1)$$

$\forall j \in [1,9]$ and

(mappy[i][j] == 0

or

vis[mappy[i][j]] == True)

and

visited[j] == false

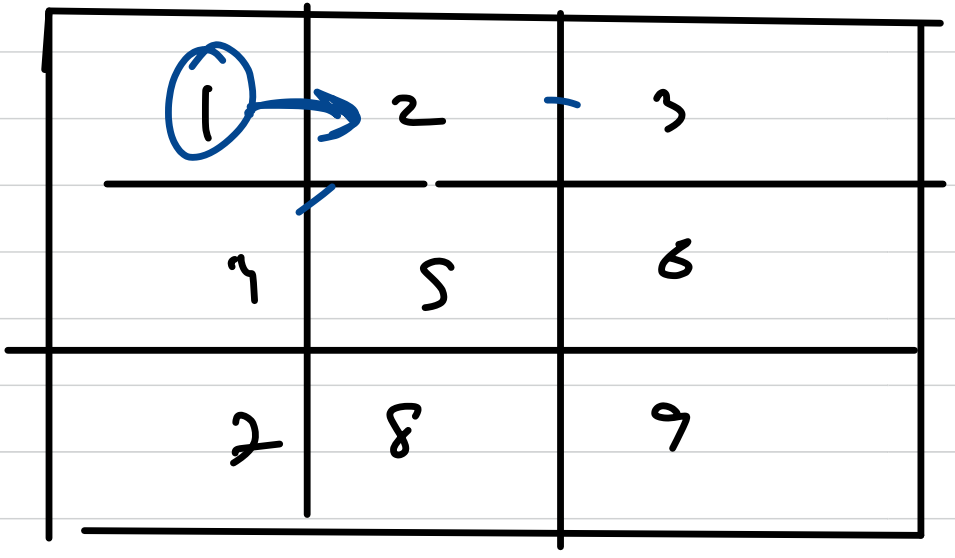
$f(visited, 1, k-1) \rightarrow x$

$f(visited, 2, k-1) \rightarrow y$

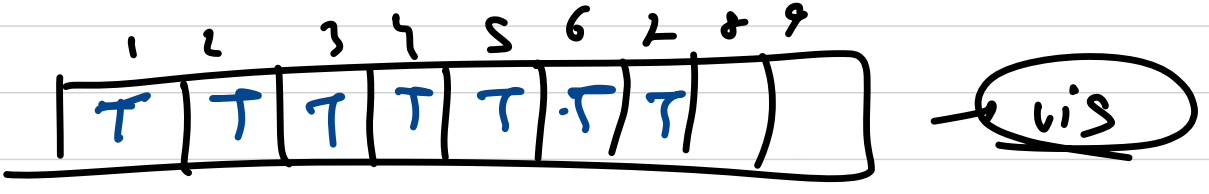
$f(visited, 5, k-1) \rightarrow z$

$4x + 4y + z$

$x = 5 \; 2 \; x \; 0$



1



Sorting Algorithms

Q. What is sorting ??

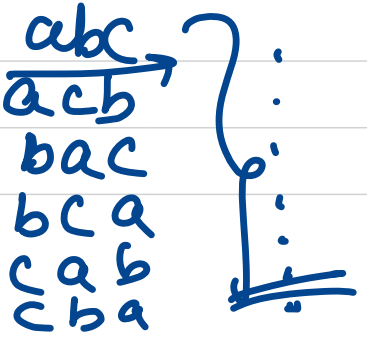
1) Less correct definition → means arranging data in inc. order.

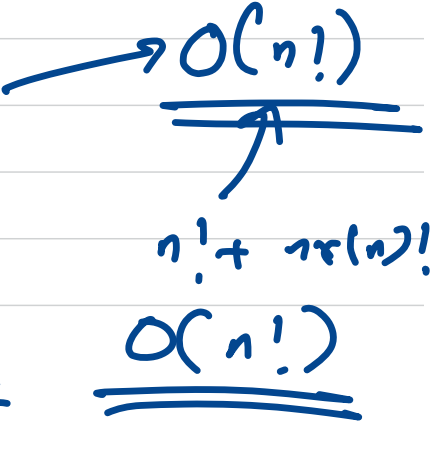
2) Correct Definition → Sorting is a process of rearrangement of data in a desired particular order.

Now, you know the definition of Sorting.
Can you devise an algorithm to sort a bunch of data in a given order. (try to brute force)

Permutation

abc →





I can find all possible permutations (rearrangements) and filter out the one which satisfies our criteria.

Selection Sort

(assume we want to sort in inc order)

Situation

1) You've an array, where initial few elements are sorted properly rest are unsorted.

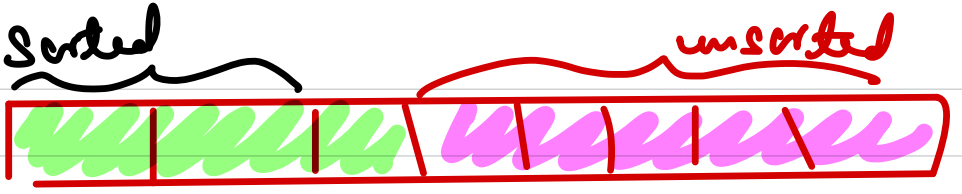
2) the largest element of the sorted part is smaller than the smallest element of the unsorted part.

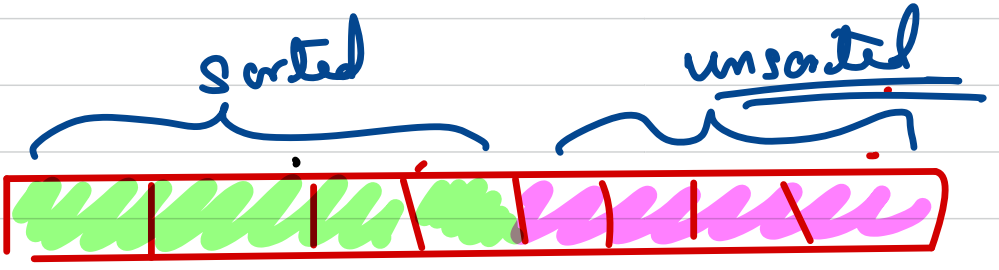
How can we grow the sorted region?

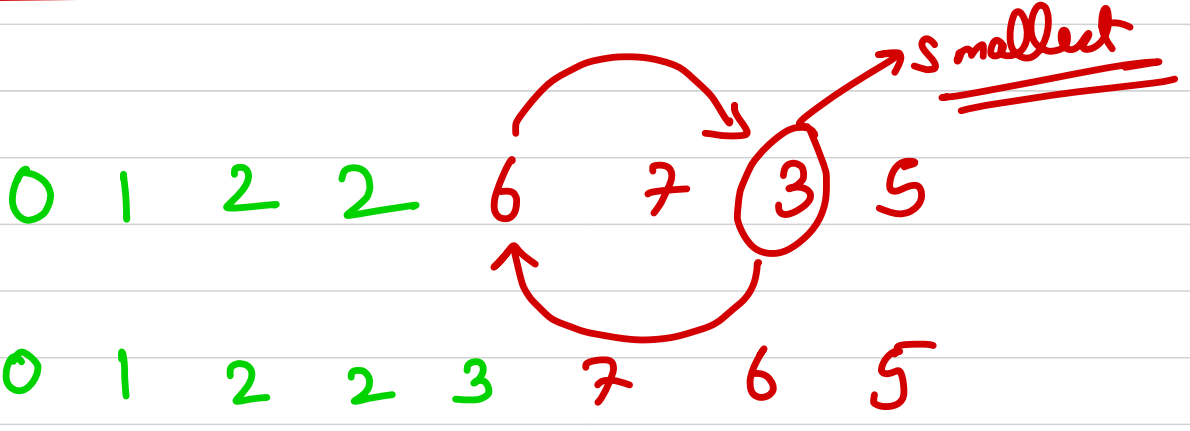
* ① find the smallest element of the unsorted region.

② And swap it with the first element of unsorted region.

Repeat ① and ② and you get Selection sort.


sorted
unsorted


sorted
unsorted


smallest
0 1 2 2 6 7 3 5
0 1 2 2 3 7 6 5

→ unsorted

→ sorted

1	2	6	7	2	5
0	1	2	3	4	5 → i.d.

start = 2

candidate_min = 4 → [start]

↳ index of min elem of unsorted region

instead of storing elem let's store index

if (arr[i] < arr[cand_min]) → True

candidate_min = i

Swap(arr, start, candidate_min)

[5, 2, 6, 7, 2, 1, 0, 3]

0 2 6 7 2 1 5 3

0 1 6 7 2 2 5 3

0 1 2 7 6 2 5 3

1st iteration → n-1 comparison
1 swap

2nd iteration → n-2 comparison
1 swap

3rd iteration → n-3 comp
1 swap

$$(n-1) + (n-2) + (n-3) + (n-4) \cdots\cdots 2 + 1$$

Sum of the first $(n-1)$ natural nos.

approx total comp $\longrightarrow$ $\frac{n \times (n-1)}{2} \rightarrow \frac{n^2}{2} - \frac{n}{2}$

Big O $\rightarrow$ $O(n^2)$ time complexity

Selection Sort → $O(n^2)$ Comparisons

$O(n)$ Swaps

Time → Time reqd for Comp + Swap

$O(n^2) + O(n)$

$\approx O(n^2)$ → worst Case

Space → $O(1)$

In the best case, array will be already sorted

Eg → 1, 12, 23, 34, 45

idx 0 1 2 3 4

i

start = 1

candidate_min = 1

1st iteration → n-1 comp

0 swap

2nd → n-2 comp

0 swap

In the best case we do only comparison & no swap

Time Complexity → $\Omega(n^2)$

Avg Case → $\Theta(n^2)$

Applications of selection sort:

Selection sort does very less swap, so in the worst case we do less swapping when compared to other algorithms. So, in places where swapping is expensive & time is enough we use selection sort. Eg → Satty fetch data from HDD.

Stability in Sorting algo.

mobile phone

Sort them based on price

Initial → Sam S20 70,000 | Iphn13 Red 80,000 | iphn13 pro Black 100000 | iphn13 Black 90,000

After sort

Case 1 → Sam S20 70,000 | iphn13 Black 90,000 | Iphn13 Red 80,000 | iphn13 pro Black 100000

→ relative order of same elects Changed

Case 2 → Sam S20 70,000 | Iphn13 Red 80,000 | iphn13 Black 90,000 | iphn13 pro Black 100000

→ relative order of same elect is same

If after Sorting the relative order of same elements remains same then algorithm is called Stable.

Q. Is selection sort stable ?.!

No

Ex

0	1	2	3
1	2''	2'	4

cand_min = 2

Bubble Sort

iter = 3

Total iter = n-2

10, 14, 27, 33, 39

i

j stops at 2

iter = 0 j = 3

iter = 1 j = 2

iter = 2 j = 1

iter = 3 j = 0

j stops at

n - iter - 2

```
if(arr[j] > arr[j+1])
    swap(arr, j, j+1)
```

In every iteration we compare adjacent elements & bubble up the biggest one.

In every iteration we will bubble up the biggest element to the last

NOTE → To do bubble sort, we can't just do opp. selection sort because it will not give us the advantage of bubble sort.

iter = 0

1 , 12 , 15 , 18 , 19

i =

flag → isSwapped = false

↳ I make it true if we swap atleast once in the iter

2 , 1 , 3 , 4 , 5

Comp → n-1
Swap → n-1
} 1

Comp → n-2
Swap → n-2
} 2

Comp → n-3
Swap → n-3
} 3

Time → $(n-1+n-1)+(n-2+n-2)+(n-3+n-3)\ldots\ldots+(1+1)$

$\Rightarrow 2\times((n-1)+(n-2)+(n-3)\ldots\ldots 1) \Rightarrow 2\times\frac{(n)(n-1)}{2}$

$$\text{time} \rightarrow \quad n(n-1) \quad = \quad n^2 - n$$

Big $O(n^2)$

Best Case → $\Omega(n)$

Worst Case → $O(n^2)$

avg Case → $\Theta(n^2)$