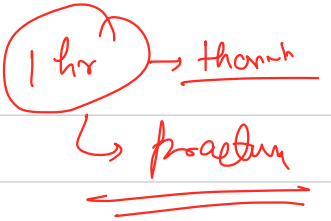
We will start at 7:35

Aman's Doubt class → 6:30 pm Zoom

↳ form → tell your doubts

Agenda → Complexity analysis of algorithms



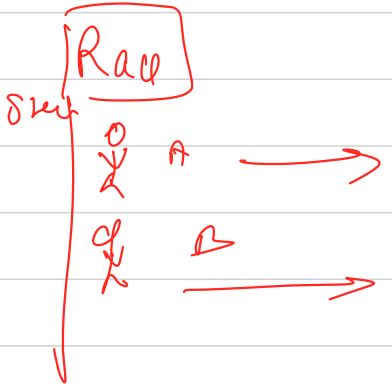
⇒ Process consumes some memory

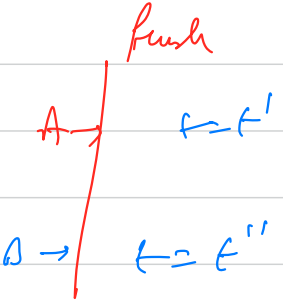
to execute a task process takes some time

Algo → memory efficient

→ Consume less time to execute the task

↪ How we can calculate how much time an algo takes to execute ??





time consumed by A → $(t' - t)$ =
" B → $(t'' - t)$

Experimental analysis

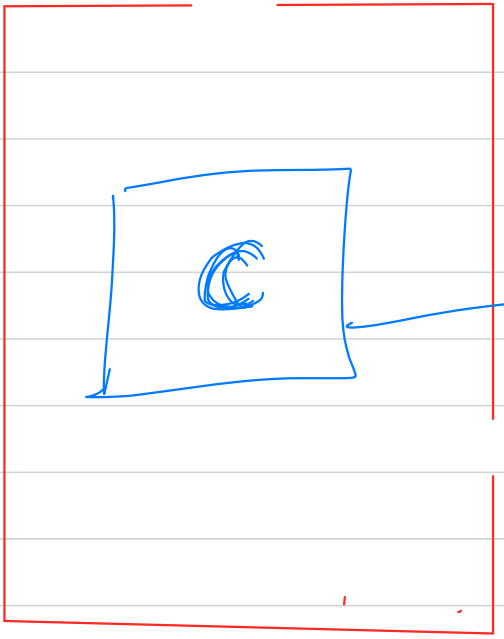
→ point at which code starts to execute → t_1

} statements of algo

→ point when the code ends → t_2

$t_2 - t_1$

Because → your computer doesn't actually executes everything parallely

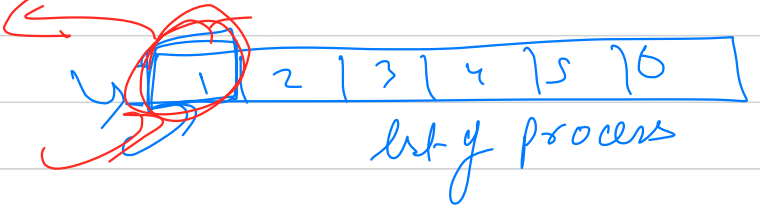


→ multiprocessing

→ multitasking

1 sec → 10^9 cycles

$t = t'$ → one process

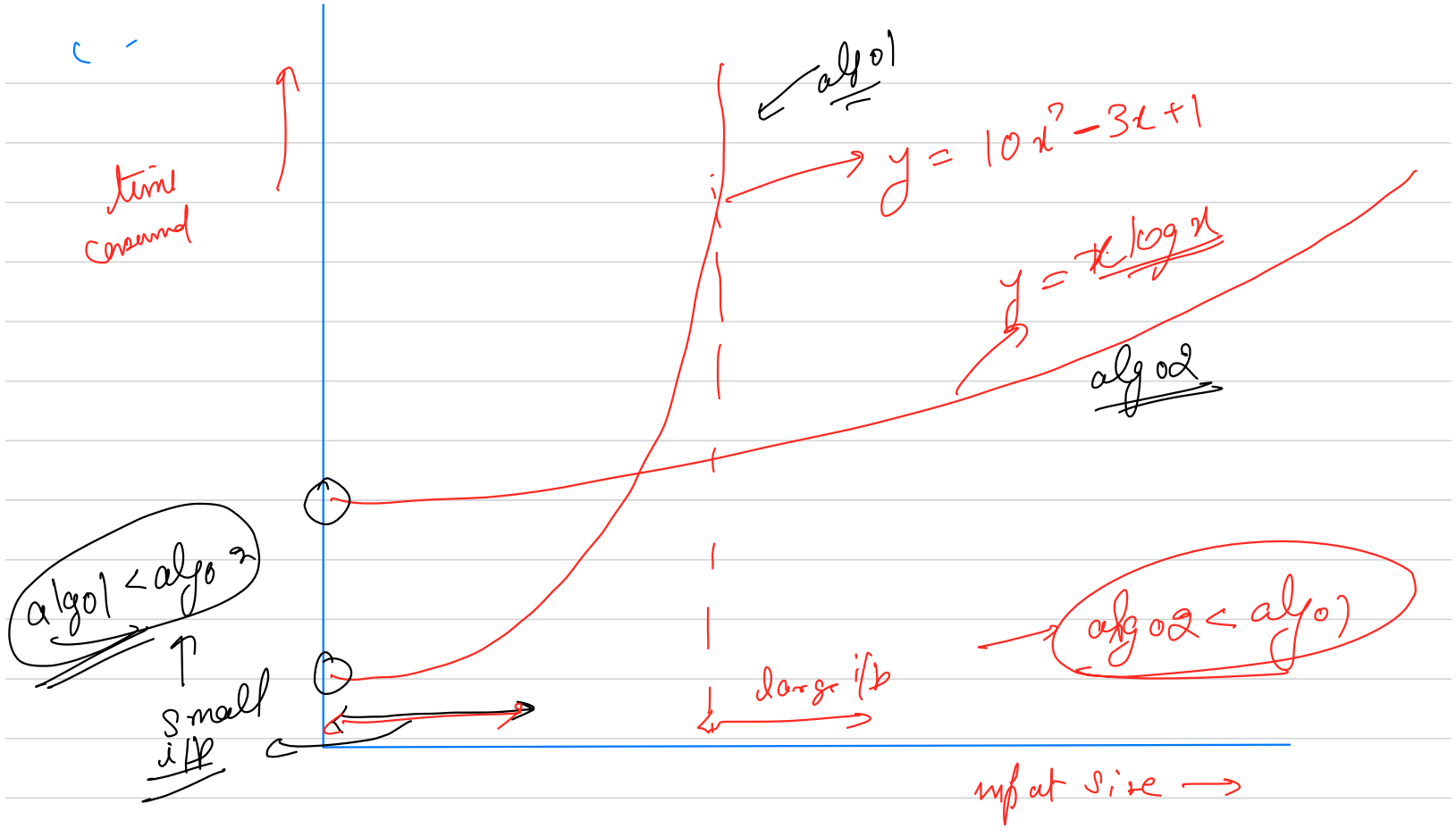


list of process

↳ microprocessor (i9, i7, i5 ....)

$t = 1$ ms ← unit

very very fast


time consumed
algo1
$y = 10x^2 - 3x + 1$
$y = x \log n$
algo2
algo1 < algo2
small i/p
large i/p
algo2 < algo1
input size

The change in execution time based on change in input is a better metric to measure.

→ Rate of growth of algorithm execution time

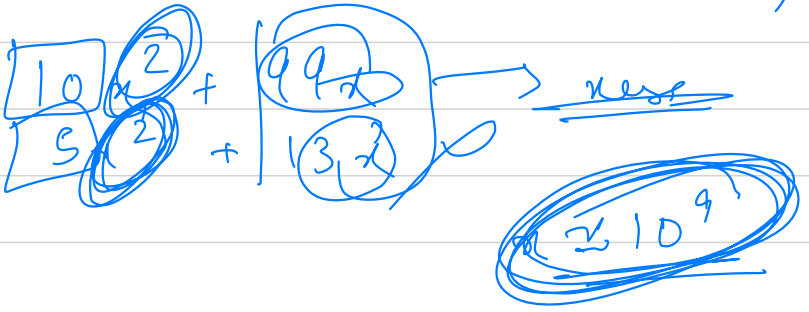
algorithm having lesser rate of growth will be better.

$$f(x) = ax^n + bx^{n-1} + cx^{n-2} \quad \cdots\cdots \quad + \text{const}$$

$$g(x) = a'x^n + b'x^{n-1} + c'x^{n-2} \quad \cdots\cdots \quad + \text{const}$$

degree $\rightarrow$ n

for a polynomial having same degree, nature of growth will be same for large value of x

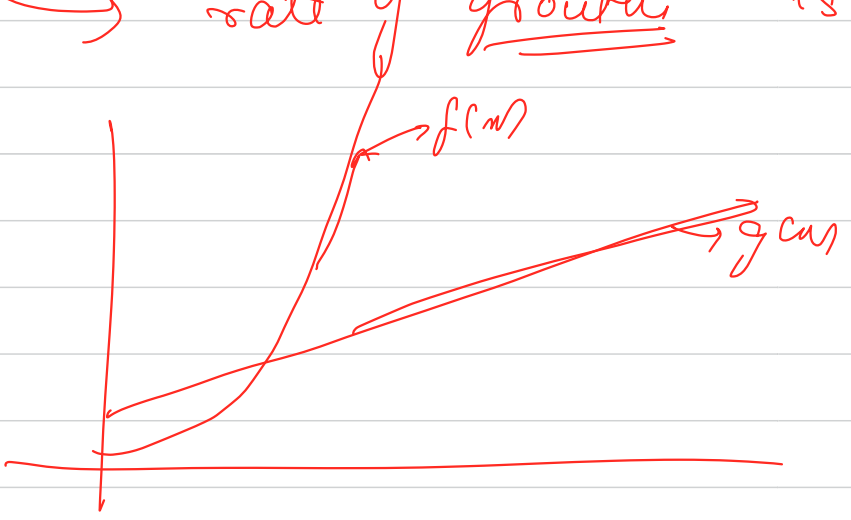


$f(n) = 3n^2 + 2n + 1$

$g(n) = 5n + 3$

$g(n)$

rate of growth is less



→ If we can approximately visualize the rate of growth in terms of the highest degree polynomial, we can compare algo.

↳ asymptotic analysis

1) Best Case → Big omega notation → $\Omega(1)$

2) Avg Case → Big theta notation → $\Theta(\sqrt{n})$

3) Worst Case → Big Oh notation → $O(\sqrt{n})$

Prime number check → 2 → n-1 → n operations

$\sqrt{n}$ operations

$n \approx 10^6$

$\sqrt{n}$ operations

2

1 operation

$n \approx 10^{16}$

n

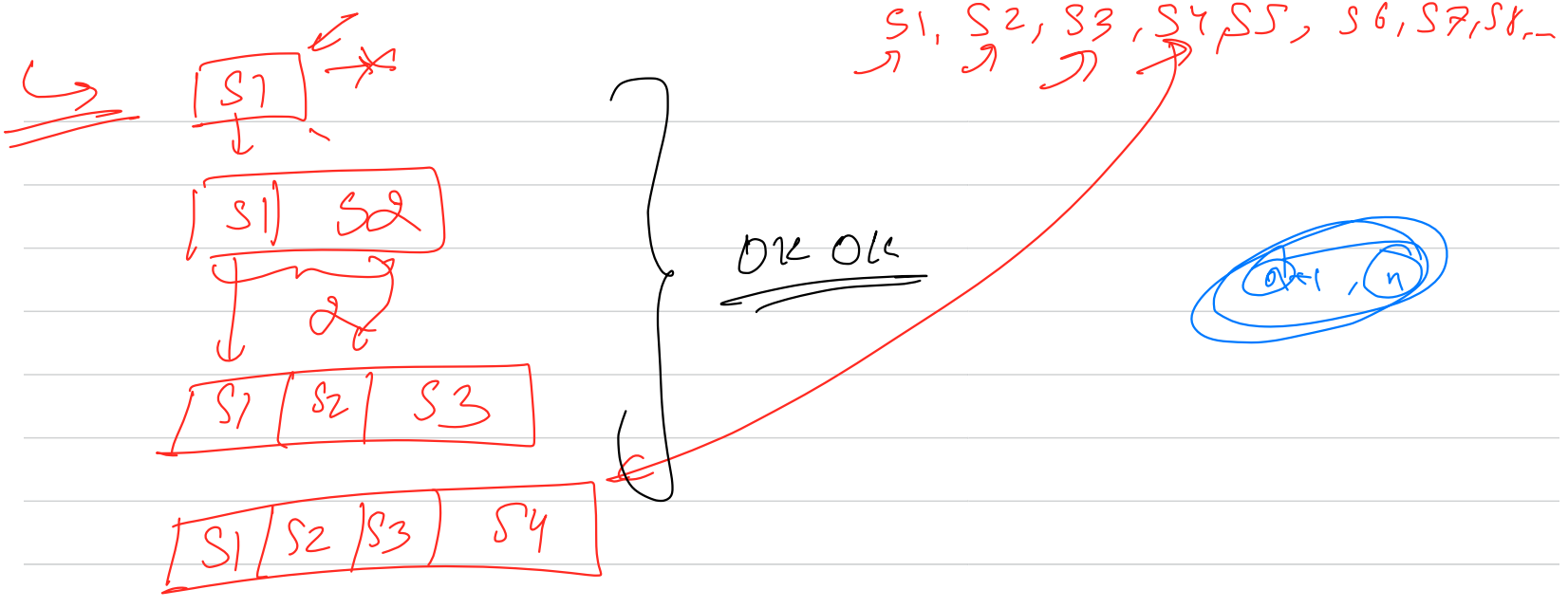
↳ const

→ implement your own insertion algorithm that inserts at the last of a list, where you can only create list of constant size.

→ You can only create a list of const size. → can never grow

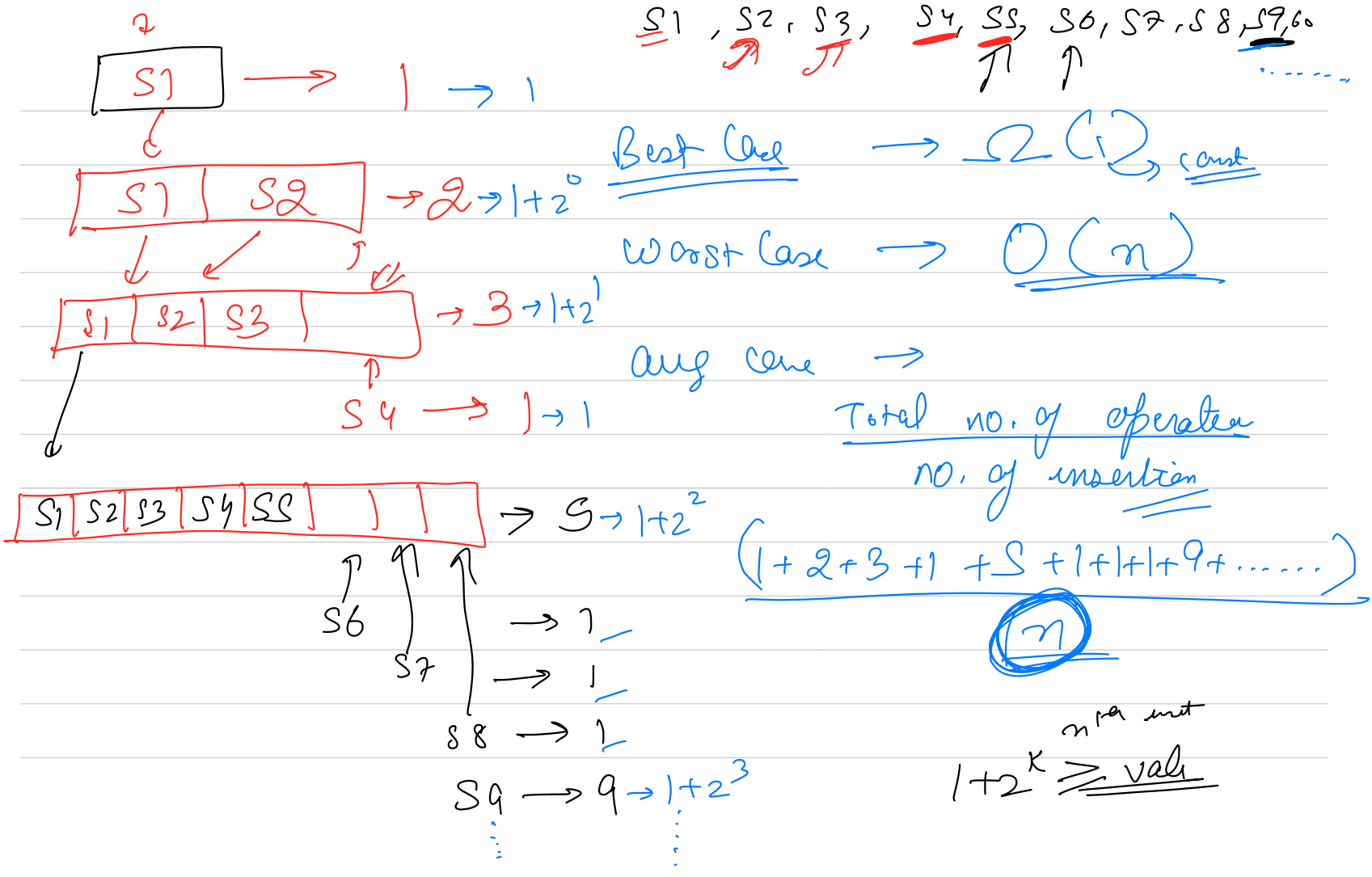
→ You cannot use append func^n

→ Instead write your own func^n called as add at last, that adds element at last.



If you have done n insertion till now, how many operations you will do for (n+1)th insertion?

$n+1$


S1 , S2 , S3 , S4 , S5 , S6 , S7 , S8 , S9 , S10 .....
2
S1
→ 1 → 1
S1 S2
→ 2 → 1+2^0
S1 S2 S3
→ 3 → 1+2^1
S4 → 1 → 1
S1 S2 S3 S4 S5
→ 5 → 1+2^2
S6
S7
S8 → 1
→ 1
→ 1
S9 → 9 → 1+2^3
Best Case → Ω(1) const
Worst Case → O(n)
avg case →
Total no. of operation
no. of insertion
(1+2+3+1+5+1+1+1+9+.....)
n
n power must
1+2^k ≥ val

$n \to 1 + 2^k$

$2^k \geq n-1$

$k = \log_2(n-1)$

$k = \log_2 n$

$1 + (1+2^0) + (1+2^1) + 1 + (1+2^2) + 1 + 1 + 1 + (1+2^3) \ldots$

n = (n times) + ($2^0 + 2^1 + 2^2 + \ldots + 2^k$, k terms)

GP

$2^{\log_2 n} = n$

$n + \frac{2^0 \times (2^{\log_2 n + 1} - 1)}{2 - 1}$

2^{k+1}

$2 \times 2^{\log_2 n}$

$2n$

$\Rightarrow n + 1 \times (n-1)$

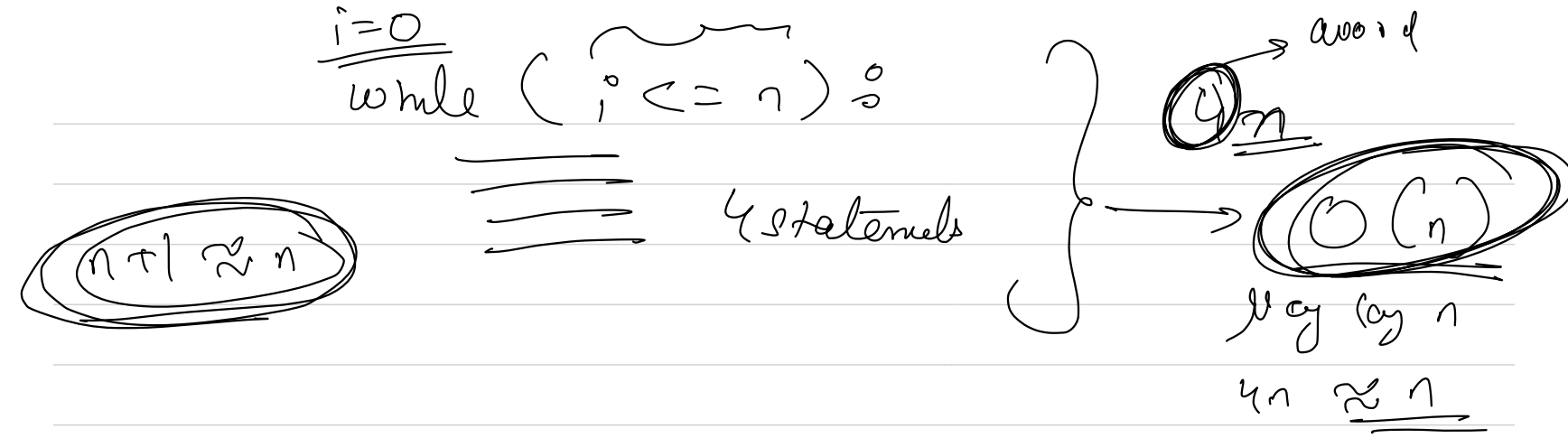
$\Rightarrow 2n - 1 \to$ const $\to$ avg $\to \Theta(1)$

O → w cast

Θ → avg

Ω → best

$\frac{2n-1}{n}$ → Load



no of operations ≈ no of times executed × no. of times they are exec

```
i = 0
while (i <= n):
    ≡≡≡  4 statements  ⟶  4n
```

```
j = 0
while (j <= m):
    ≡≡≡  4 statements  ⟶  4m
```

$$4n + 4m$$

$$\approx O(n+m)$$

nested loop

Time complexity

```
i=0
while (i<=n):                    n
    j=0
    while (j<=n):                n
        print("Hello")

k=0
while (k<=m):                    m
    print("World")
```

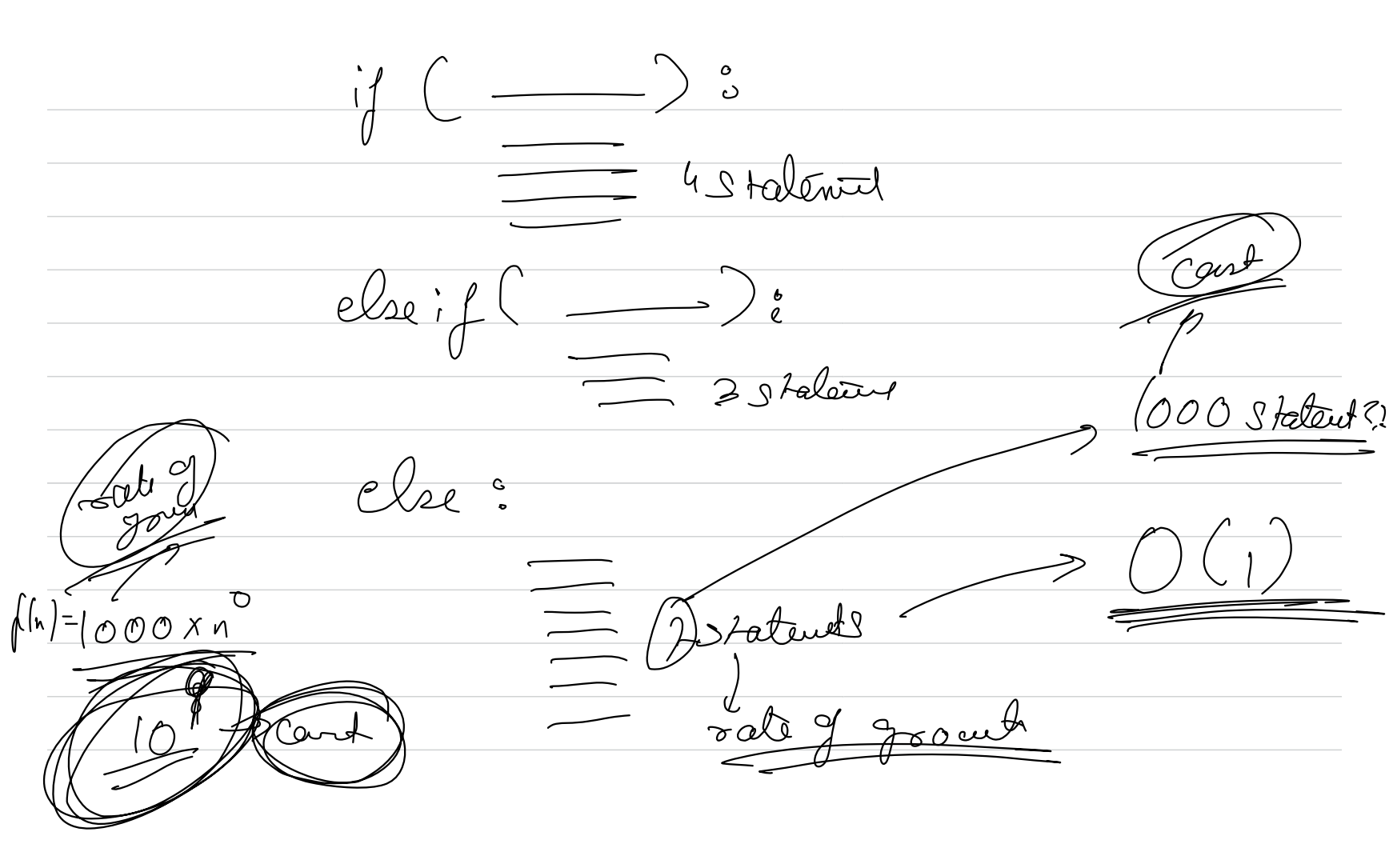
$O(n \times (n+m))$

$\Omega(n \times (n+m))$

$\Theta(n \times (n+m))$

We are more concerned about avg & worst case &

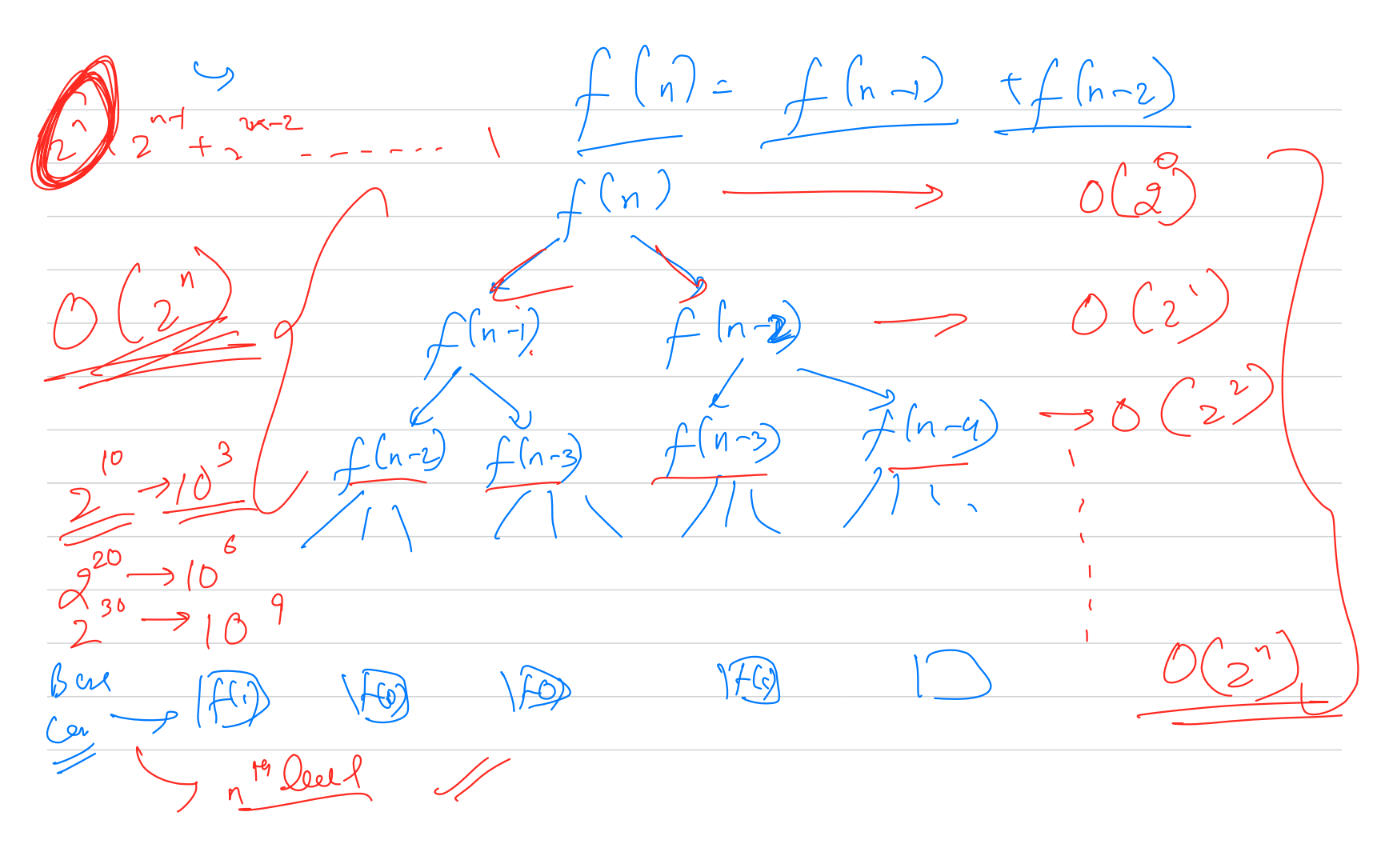
mainly worst case

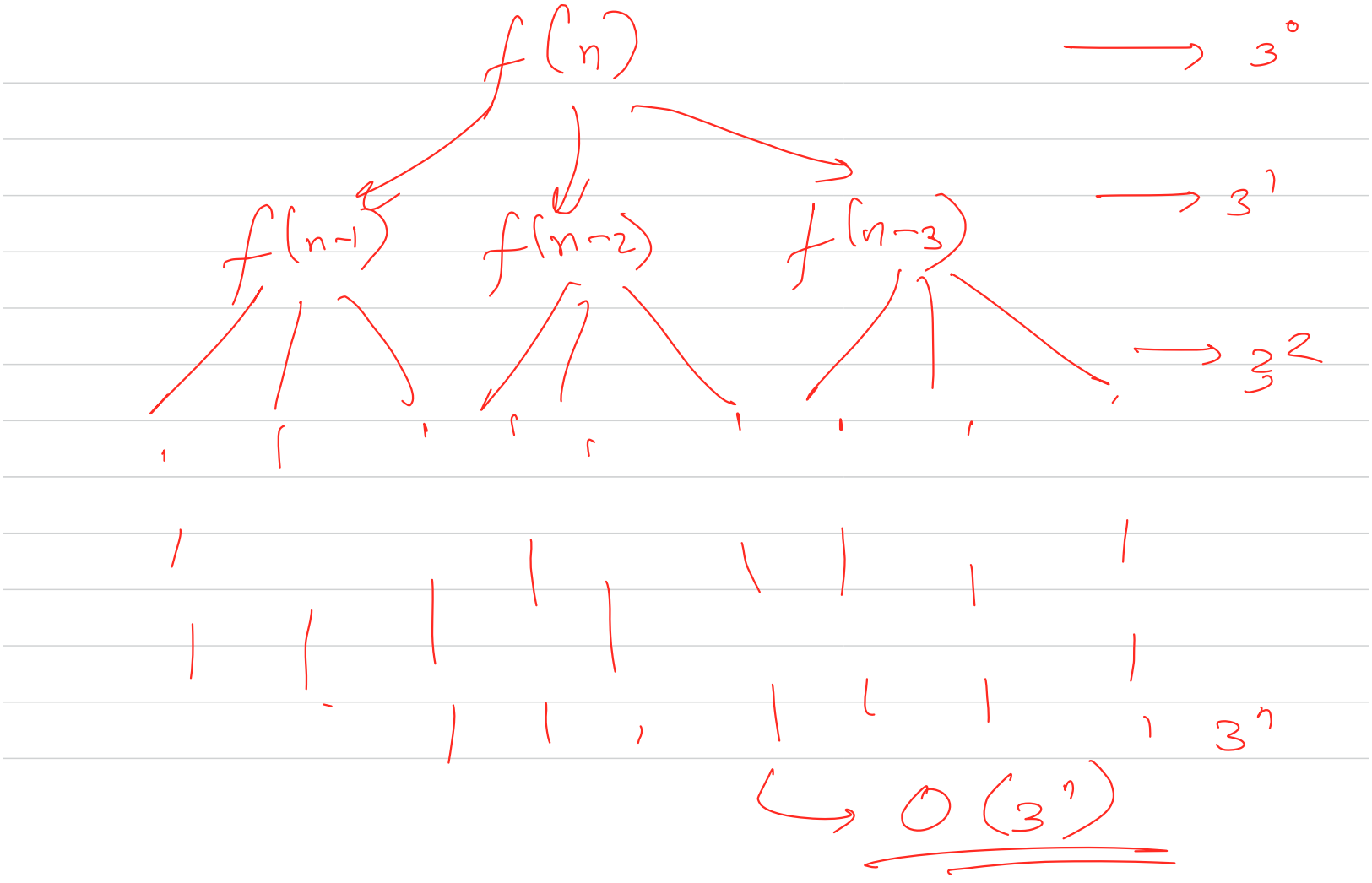

if (———) :
4 statement
else if (———) :
2 statement
else :
2 statements
rate of growth
Constant
1000 statements
O(1)
rate of growth
f(n) = 1000 × n^0
10^0
Constant

$O(n^2)$

$i=1$ $i=2$ $i=3$

$1 + 2 + 3 - - - - - -$


f(n) = f(n-1) + f(n-2)
f(n)
O(2^n)
f(n-1)
f(n-2)
O(2^1)
O(2^2)
f(n-2)
f(n-3)
f(n-3)
f(n-4)
2^10 → 10^3
2^20 → 10^6
2^30 → 10^9
O(2^n)
n^th level


f(n)
f(n-1)
f(n-2)
f(n-3)
3^0
3^1
3^2
3^n
O(3^n)