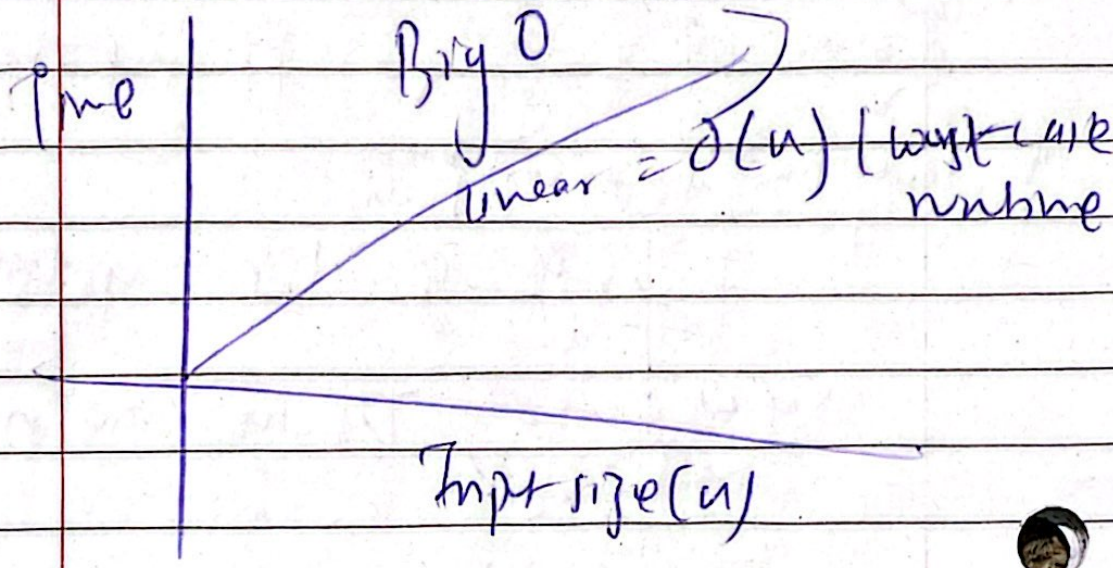
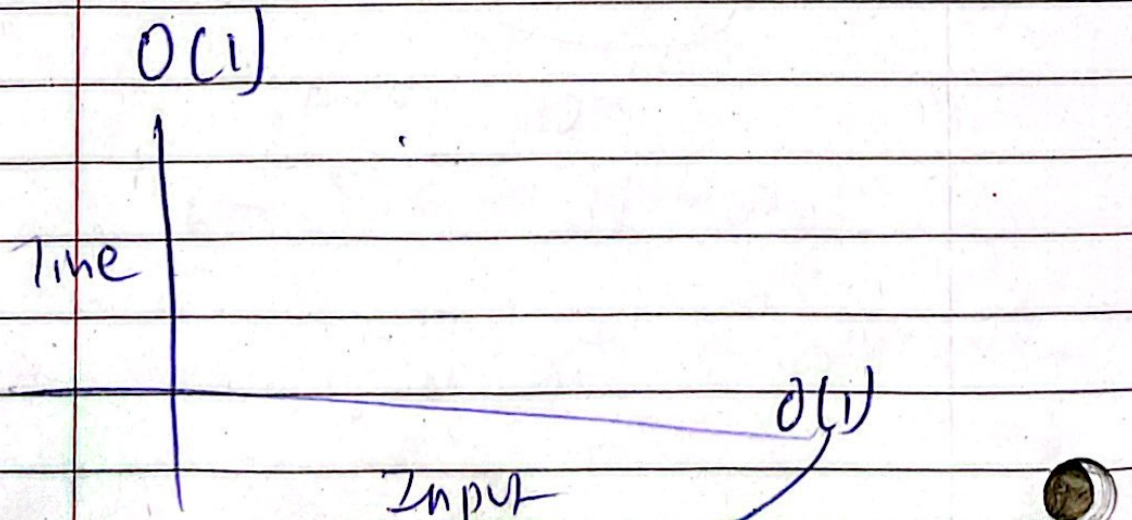
Big O time complexity

- Amount of time taken for an algorithm to execute



we don't count constant like n+1 or n/2

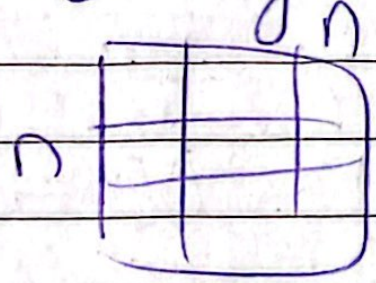


1 constant we care about no matter size of input, time is same

$O(n^2)$

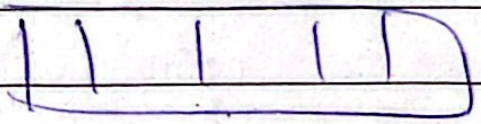
→ Usually nested loop

→ 2d array



[[...], [?.

@ $= n^2$



→ n, go through time

$2 \times n = n$

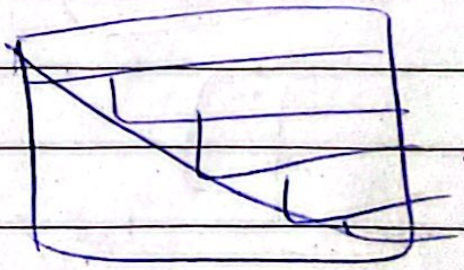
but if you go through it

n times

$n \times n = n^2$

or, $n \times (n-1) \times (n-2)$

1st index, 2nd index

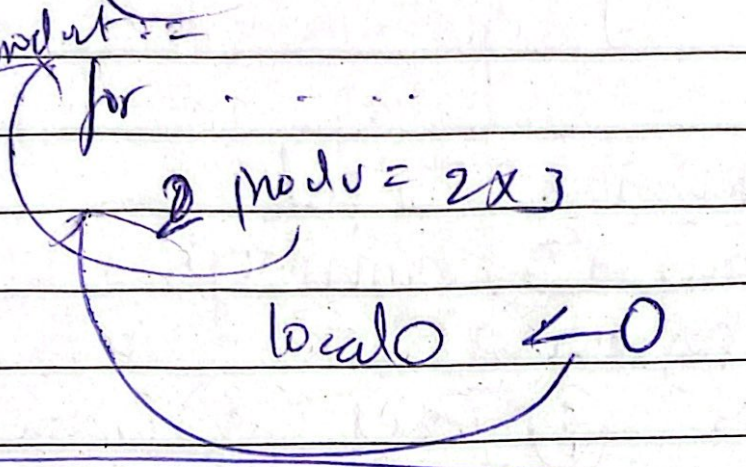


→ $\frac{n^2}{2}$?!

we don't care about 2 (constant, so $O(n^2)$

product :=

(2, 3, -2, 4)



Dynamic programming:

→ optimization over recursion

→ ~~can~~ Any recursive solution can be optimized with dynamic programming

→ Idea is to store result of subproblems so that we do not have to recompute them when we need it later

→ Remember your past.

- If a given problem can be broken down into smaller problem & again for those smaller problems & in prob, we see some overlapping problems.

→ Try : Lot many soln pick the best one.

→ Solved using iteration

→ Optimization

Principle of optimality

problem can be solved By taking best & soft decision

fibonacci

0, 1, 1, 2, 3, 5, 8, 13

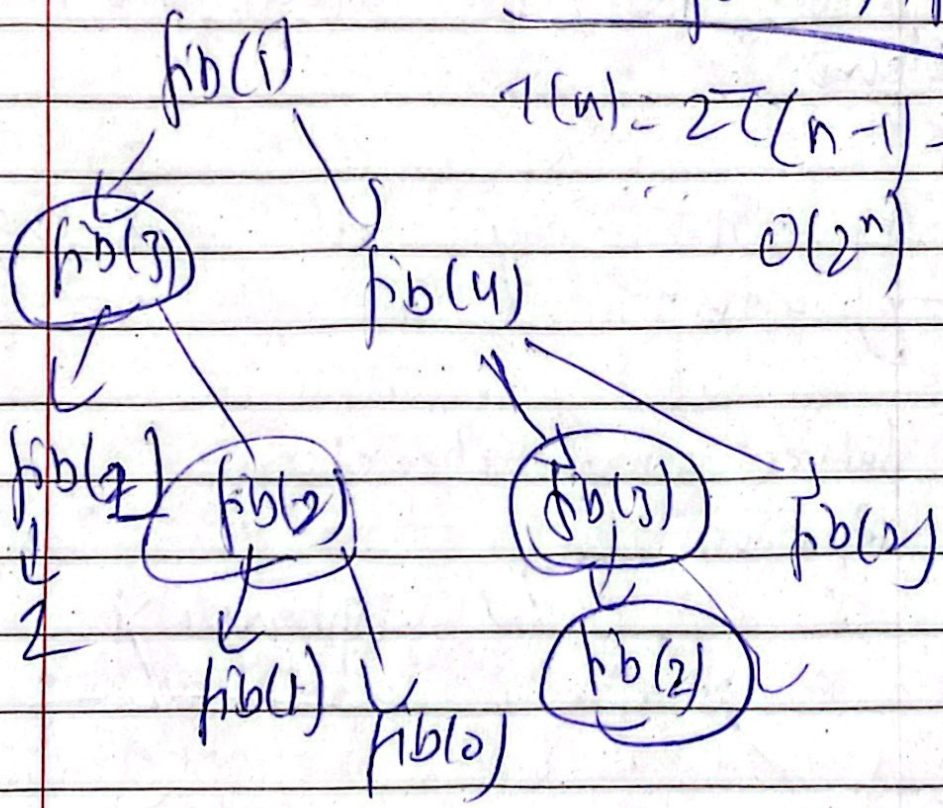
1 2 3 4 5 6 7

normal

```
int fib(int n)
{
if (n<=1)
return n;
return (fib n-2) + fib n-1)
```

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

$O(2^n)$



Tracing tree of that problem

15 calls

optimization

f

-1	-1	-1	-1	-1	-1
0	1	2	3	4	5

f(5) = -2 we don't know

f(3) fib4

fib(1)

f[0] = 0

f(1)

f[0] = 1

fib(1) already known, so just append in array

f [0|1|1|2|3|5]

fib(n) = n+1 calls O(n)

Memoization: storing result of functions, not calling again

Top Down Approach

Tabulation

```
fib(n)
int fib(int n)
    if n<=1
        return n;
    f[0]=0; f[1]=1;
    for(int i=2; i<=n; i++)
        f[i] = f[i-2] + f[i-1];
    return f[n]
```

Adding previous terms

f	0	1	1	2	3	...	f[n]

Table is generated bottom up approach because we start from 0

Principles

(1) Characterize recursive soln,

(2) build a mathematical model for soln

(3) Recursively define value of optimal soln

(3) Using bottom up approach, compute value of optimal soln for each possible subproblems

(4) Construct optimal soln for original problem using info computed in previous step

Problems

Any that require maximizing or minimizing certain quantities or counting

- Overlapping certain problems