Data structure and Algorithms

Course

Created by : Israa Abdelghany
LinkedIn : www.linkedin.com/in/israa-abdelghany-4872b0222
GitHub : https://github.com/IsraaAbdelghany9

Day 3

Review last lecture

➔ time Complexity :-

- Concerns about the relation between # of inputs with time
- relation between n & number of iterations (time)

Example :-

```
for (i=1 ; i<n ; i=*2)
  {
     statment;
  }
```

	n	no. of iter.
2^1	2	1
2^2	4	2
2^3	8	3
2^4	16	4

$\therefore \log_2 n$

```
for (i=1 ; i*i<n ; i++)
  {
     statement ;
  }
```

n	iter.
4	1
8	2
16	3

$\log_2(n) - 1 \equiv \log_2 n$

```
for (i=1 ; i*i<n ; i*2)
  {
     statement ;
  }
```

n	iter.
2	1
4	1
8	1, 1
16	1 , 1
32	1, 1 , 1
64	1,1,1

$\log_2(n)$

if i starts with 1

1	5
2	5
3	5
4	5
5	X

→ 20

$(n-1)\,n$

$n^2 - n$

$O(n^2)$ also

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

i	iter.
1	5
2	5
3	5
4	5
5	5
6	X

iter

j

i

1

2

4

---

assume n = 5

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

i	iteration
0	0
1	1
2	2
3	3
4	4
5	X

time is a series.

$$t = \frac{n * (n+1)}{2}$$

$$= \frac{n^2 + n}{2}$$

$O(n^2)$

```
for (i=1; i<n; i*=2)
{
    for (j=0; j<i; j++)
    {
        statement
    }
}
```

n=8

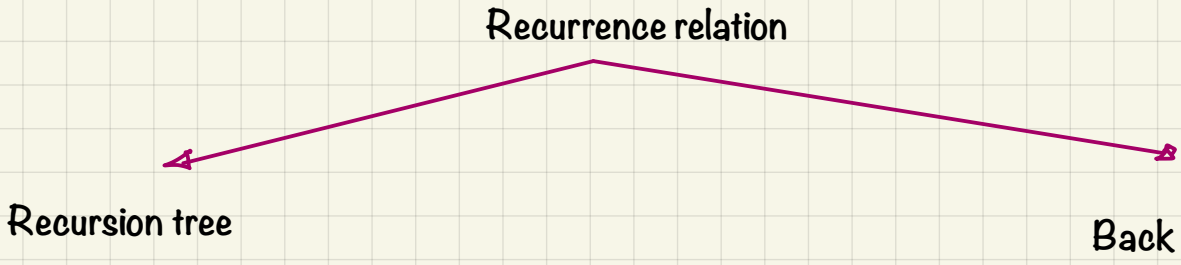
i	iter
1	1
2	2
4	4

$1+2+4+8+\ldots\ldots$

$1 \quad 2 \quad 4 \quad 8 \quad \ldots\ldots$

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

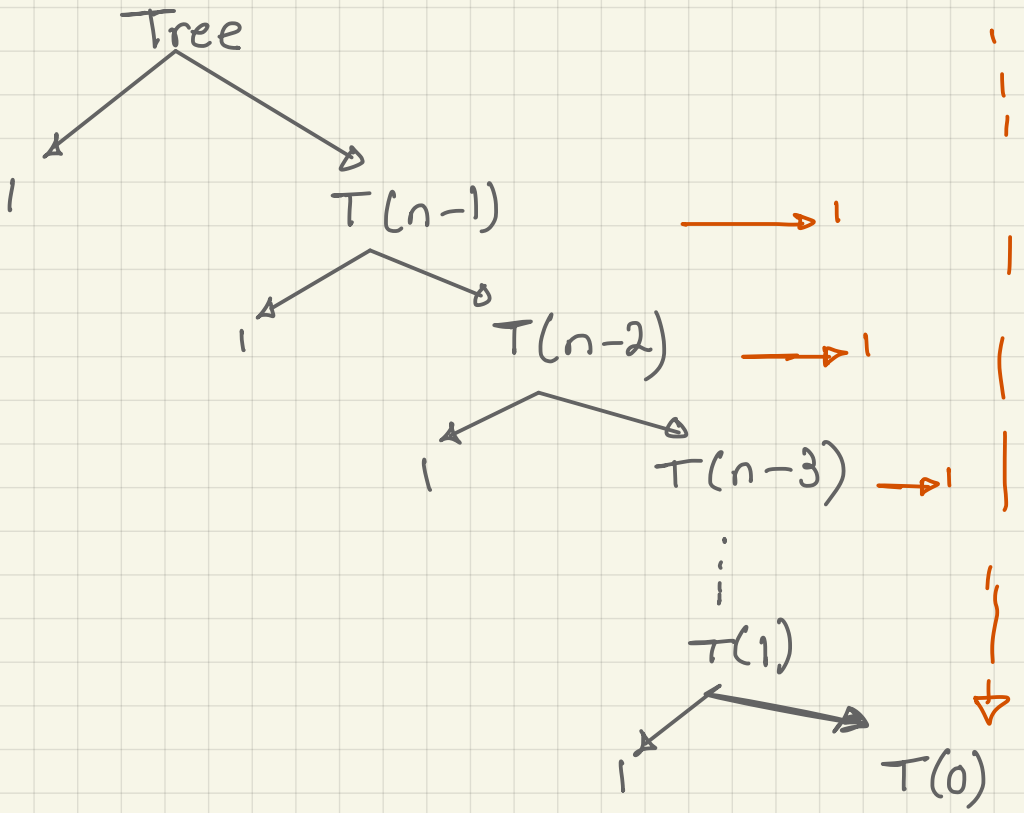
$O(2^{\log n})$



Ex1

```
fl (int n)  ⟶ T(n)
{
    if (n > 0)
    {
        print (n)  ⟶ 1
        fl (n-1)   ⟶ T(n-1)
    }
}
```

$$T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1)+1 & ; n > 0 \end{cases}$$



$T(n) = n+1$

$O(n)$

Back substitution:-

eq① ← $T(n) = T(n-1) + 1$ ⟶ I

$T(n-1) = T(n-2) + 1$ ⟶ II

$T(n-2) = T(n-3) + 1$ ⟶ III

$T(n) = (T(n-2) + 1) + 1$ ⟶ IV

eq② ← $= T(n-2) + 2$

$T(n) = (T(n-3) + 1) + 2$

eq③ ← $= T(n-3) + 3$

∴ $T(n) = T(n-k) + k$ ⟹ we took into consideration eq (1, 2, 3) to recognize the pattern

$n - k = 0$

$k = n$

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

$= n$

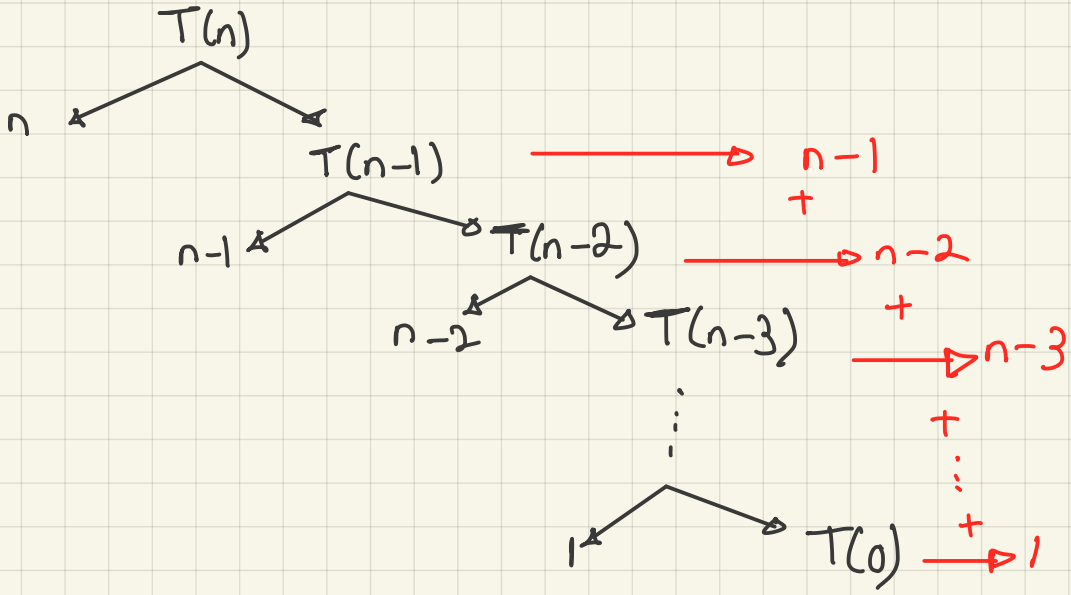
∴ $O(n)$

Ex 2

```
fl (int n)  ---> T(n)
{
    if (n > 0)
    {
        for (i = 0; i < n; i++)
        {
            print (n)
        }
    }
    fl (n-1)  ---> T(n-1)
}
```

n times

$$T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1) + n & ; n > 0 \end{cases}$$



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

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

$$\Rightarrow O(n^2)$$

Back substitution:-

$$T(n) = T(n-1) + n$$
$$T(n-1) = T(n-2) + (n-1)$$
$$T(n-2) = T(n-3) + (n-2)$$

sub2 in 1

$$T(n) = [\, T(n-2) + (n-1) \,] + n$$
$$= T(n-2) + (n-1) + n$$

sub3 in 4

$$T(n) = [\, T(n-3) + (n-2) \,] + (n-1) + n$$
$$= T(n-3) + (n-2) + (n-1) + n$$

So pattern is

$$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \cdots + n$$

$n - k = 0$

So $n = k$

$$T(n) = T(n-n) + (n-(n-1)) + (n-(n-2)) + \cdots n$$
$$T(n) = T(0) + 1 + 2 + \cdots + n$$

Same pattern as tree

Master theory for decreasing fun

Using Back substitution

$T(n) = T(n-1) + 1$ ①

$T(n-1) = T(n-2) + 1$ ②

$T(n-1) = T(n-3) + 1$ ③

② in ①

$T(n) = [T(n-2) + 1] + 1 = T(n-2) + 2$ ④

③ in ④

$T(n) = [T(n-3) + 1] + 1 = T(n-3) + 3$

from those eqns we find that

$T(n) = T(n-k) + k$ #

at $T(0)$ $k = n$ [Last case]

$T(n) = T(n-n) + n = T(0) + n = \boxed{n+1}$

same as recursion Tree $O(n)$ complexity

Master theorem for decreasing function

$T(n) = a\,T(n-b) + f(n)$ $\quad a, b > 0$, $f(n) = O(n^k)$

$O(f(n)) = O(n^k)$ $\quad a < 1$

$O(n * f(n)) = O(n^{k+1})$ $\quad a = 1$

$O(a^{n/b} * f(n)) = O(a^{n/b} * n^k)$ $\quad a > 1$

ex:

- $T(n) = T(n-1) + 1 \Rightarrow O(n)$
- $T(n) = T(n-1) + n \Rightarrow O(n^{1+1}) = O(n^2)$
- $T(n) = T(n-1) + \log n \Rightarrow O(n \log n)$
- $T(n) = 2T(n-1) + 1 \Rightarrow O(2^n)$
- $T(n) = 2T(n-1) + n \Rightarrow O(n 2^n)$
- $T(n) = 2T(n-2) + 1 \Rightarrow O(2^{n/2})$

Master Theorem for Decreasing function

$T(n) = a\,T(n-b) + f(n)$ $\quad a > 0, b > 0$, $f(n) = O(n^k) \Rightarrow k \geq 0$

$a < 1 \Rightarrow O(n^k)$

$a = 1 \Rightarrow O(n * n^k)$

$a > 1 \Rightarrow O(a^{n/b} * n^k)$

$n * 1$ $\quad n * n$ $\quad n * \log$

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

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

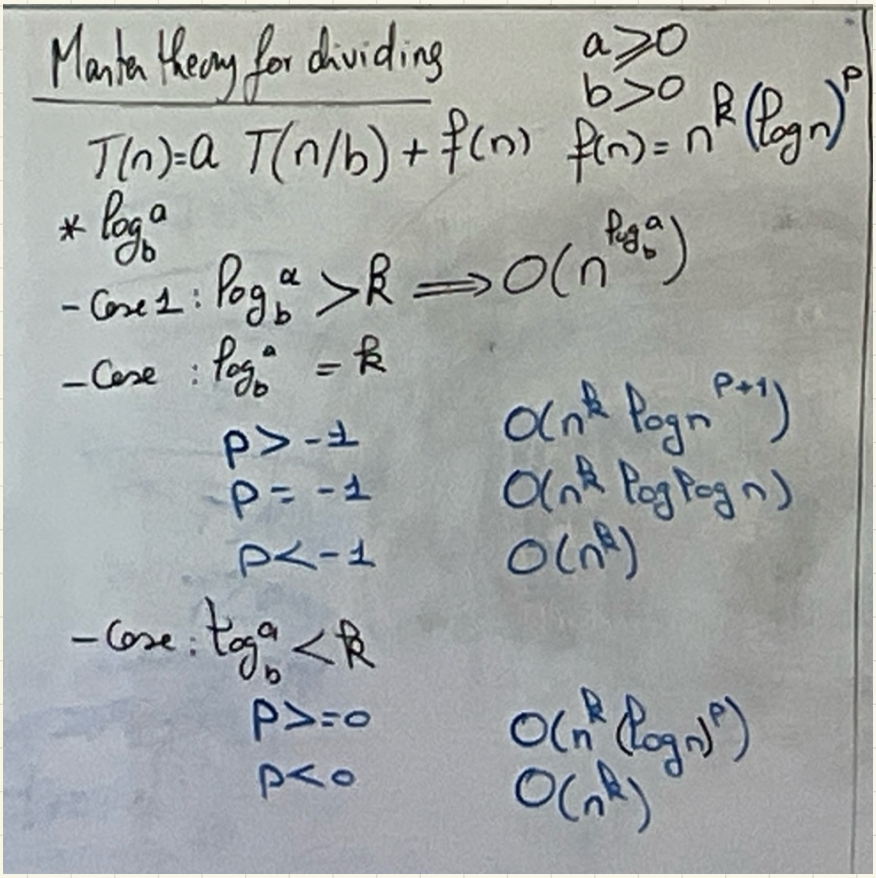
$T(n) = T(n-1) + \log n$ $O(n \log n)$

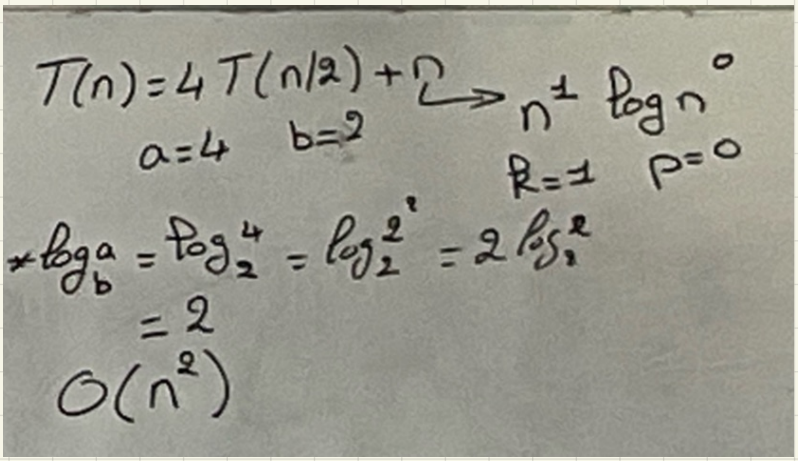
$T(n) = 2T(n-1) + 1$ $O(2^n)$

$T(n) = 2T(n-2) + 1$ $O(2^{n/2})$

$T(n) = 3T(n-1) + 1$ $O(3^n)$

Master theory for decreasing fun





$T(n) = 4\,T(n/2) + n^2 \log n$

$a=4$, $b=2$, $k=2$, $p=1$

$\log_b a = \log_2 4 = 2 = k$ Case (2 -1)

$O(n^2 (\log n)^2)$

$T(n) = 2\,T(n/2) + n/\log n$

$a=2$, $b=2$,

$\log_a b = \log_2 2 = 1 = k$ Case(2-2)

$O(n \log \log n)$

$T(n) = 2T(n/2) + n/\log n^2$

$= k$ (case 2-3)

$O(n)$

$T(n) = T(n/2) + n^2 \longrightarrow n^2 (\log n)^0$

$a = 1$, $b = 2$, $k = 2$, $p = 0$

$\log_2 1 = 0 < k$ Case 3-1

$O(n^2)$

$T(n) = 4(T/2) + n^3/\log n$

$a = 4$, $b = 2$, $k = 3$, $P = -1$

$\log_b a = \log_2 4 = 2 < k$ Case (3-2)

$O(n^3)$

Greedy Technique

- optimal local solution ⇒ hope it will lead to Global solution

- optimal solution seeks for → min / or → max

3 steps:-

- create candidate set :-
  - ↳ all Candidates
- Feasibility set :-
  - ↳ define candidates meets specific requirements
- optimal solution :-
  - ↳ find best solution

→ Greedy :- is about the local best solution but overall (global) best solution may not be selected because I'm Looking on the best choice in each solution

knapsack :- 'an application of Greedy'

types :- ① divisible ⇒ Greedy

② non divisible ⇒ dynamic Prog.

- calculate value/weight

Job sequencing with deadline :-

- Sort projects according to profit
- $d_i \Rightarrow$ deadline $\quad 1 < r \leq d_i \quad [r-1, r]$

Greedy technique :-

- Advantages :-
  - → it's easy to implement
  - → need less computing resources
  - → it's much faster in executing
- Disadvantages :-
  - → don't always reach the global optimum solution
  - → Even with the correct algorithm it's hard to prove why it is correct

Knapsack

Example:

Given the Following object with profit and weight

Object	1	2	3	4	5	6	7
Profit	5	10	15	7	8	9	4
Weight	1	3	5	4	1	3	2

Knapsack can hold max.W = 15

We need to select the items to get Max profit

we can solve the problem by 3 approaches

① max profit first

② min weight first

③ value / weight higher first

approach 1 :-

object	profit	weight	Remaining weight	total profit
3	15	5	15-5=10	15
2	10	3	10-3=7	15+10=25
6	9	3	7-3=4	25+9=34
5	8	1	4-1=3	34+8=42
4	7	4	3-3=0	$42+\frac{3}{4}*7$ =47.25

approach 2 :-

object	profit	weight	Remaining weight	total profit
1	5	1	15-1=14	5
5	8	1	14-1=13	5+8=13
7	4	2	13-2=11	13+4=17
6	9	3	11-3=8	17+9=26
2	10	3	8-3=5	26+10=36
4	7	4	5-4=1	36+7=43
3	15	5	1-1=0	43+$\frac{1}{5}$x15 =46

Approach 3:-

object	1	2	3	4	5	6	7
profit	5	10	15	7	8	9	4
weight	1	3	5	4	1	3	2
p/w	5	3-33	3	1-75	8	3	2

object	profit	weight	Remaining weight	total profit
5	8	1	15-1=14	8
1	5	1	14-1=13	13
2	10	3	13-3=10	23
3	15	5	10-5=5	38
6	9	3	5-3=2	47
7	4	2	2-2=0	51

The best approach

Job sequencing with deadline :-

Job sequencing with deadline

Problem:

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline.

It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1.

How to maximize total profit if only one job can be scheduled at a time.

Job sequencing with deadline

Solution:

The solution begins with sorting all the jobs by their profit in a decreasing order, then move along all the jobs and allocate the job in the appropriate slot depending on its deadline.

If the appropriate slot isn't empty, then search for a slot prior to it. If there is no empty slot prior to it, then reject the Job.

The Max profit, will be the sum of all profit of the Jobs allocated in slots

Job sequencing with deadline

1. Sort pi into decreasing order where pi is the profit
2. Add the next job I to the solution set if I can be completed by its deadline. Assign I to time slot [r-1, r], where r is the largest integer such that $1 \leq r \leq$ di and [r-1, r] is free
3. Stop if all jobs are examined. Otherwise, go to step 2

```
JobSequence(Set J, int n)
{
        J = {1}
        For I = 2 to n
        If(all Jobs in J ∪ {I} can be completed by their deadlines)
                J = J ∪ {I}
}
```

Example:-

Job	J1	J2	J3	J4	J5	J6	J7
Deadline	2	4	3	2	3	1	1
Profit	40	15	60	20	10	45	55

step 1 :- sort by the profit (descending order)

Job	J3	J7	J6	J1	J4	J2	J5
Deadline	3	1	1	2	2	4	3
Profit	60	55	45	40	20	15	10

step 2 :- maximum deadline = 4 at J2

Step3: First Job "J3" has dead line 3, it can be allocated at [0, 1], [1, 2], [2,3]

We select the last slot [2, 3]

i	Job	Deadline	Profit	Consider	Assign
1	J3	3	60	[0, 1], [1, 2], [2, 3]	[2, 3]
2	J7	1	55	[0, 1]	[0, 1]
3	J6	1	45	[0, 1] "Not Empty"	
4	J1	2	40	[0, 1], [1, 2]	[1, 2]
5	J4	2	20	[0, 1], [1, 2] "Not Empty"	
6	J2	4	15	[0, 1], [1, 2], [2, 3], [3, 4]	[3, 4]
7	J5	3	10	[0, 1], [1, 2], [2, 3] "Not Empty"	

Step 4: The Job Sequence for Max profit = {J3, J7, J1, J2}

And executed in the following = {J7, J1, J3, J2}

"Dynamic programming"

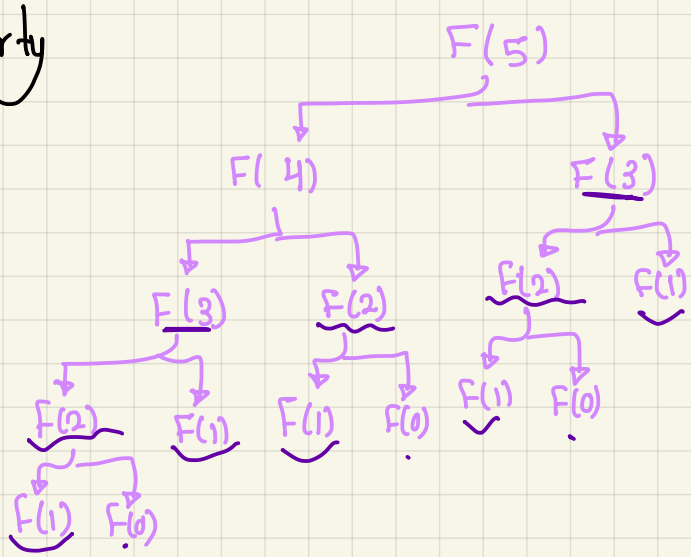
→ optimal solution

Characteristics :-

- over Lapping Sub_problems
- optimal structure property

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

Febunacchi Numer

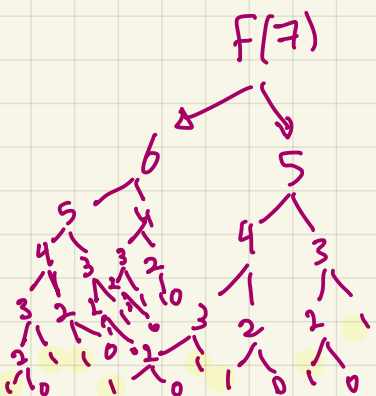


Dynamic programming Methods:-

1. Top_down with Memoization
2. Bottom_Up with Tabulation F(0) → F(1) → F(2) ···· etc

- the opposite of Memorization
- filling up n_dimensional table based on the table the solution is computed

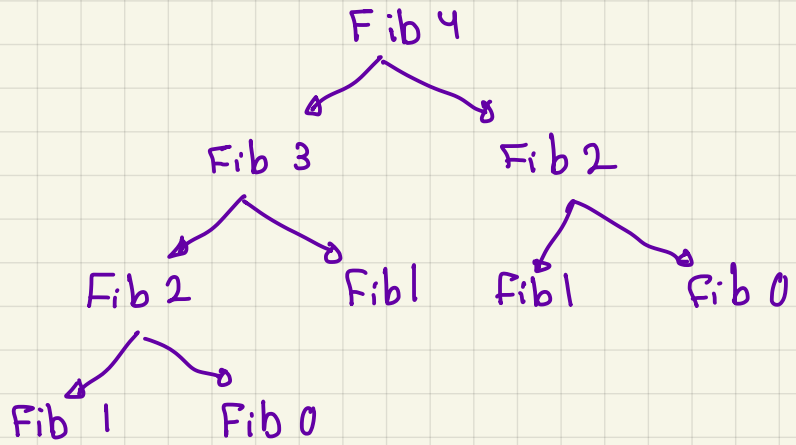
$$Fib(n) = \begin{cases} n & n<1 \\ Fib(n-2) + Fib(n-1) & n>1 \end{cases}$$



characteristics of dynamic programming:-

1. overlapping sub-problems

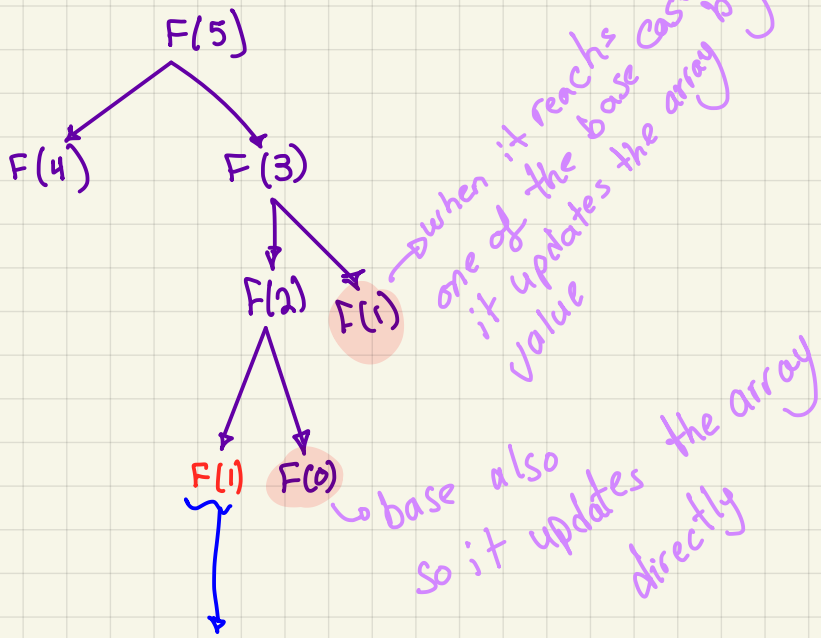
Example is Fibonacci numbers



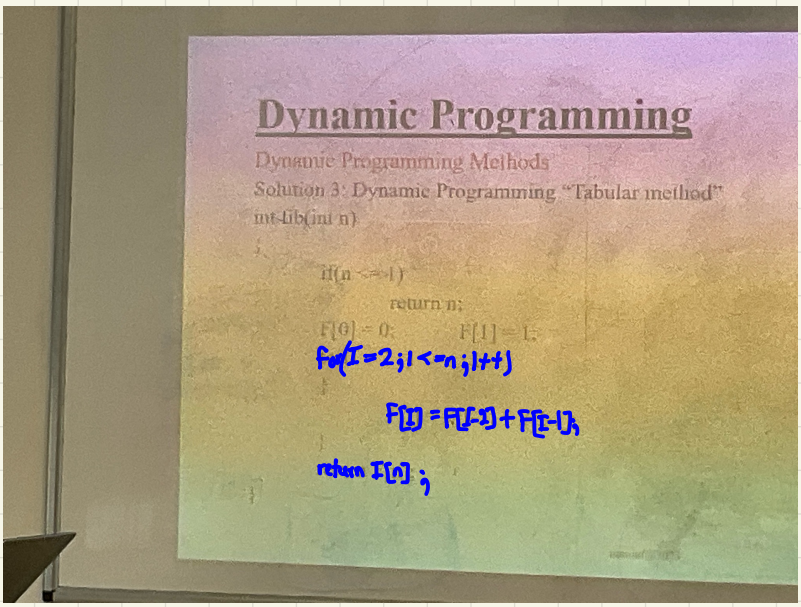
2. optimal substructure property (principle of optimality)

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

memorize method:-



if I wrote it with red that means I didn't call the function ⇒ it's already saved in the array

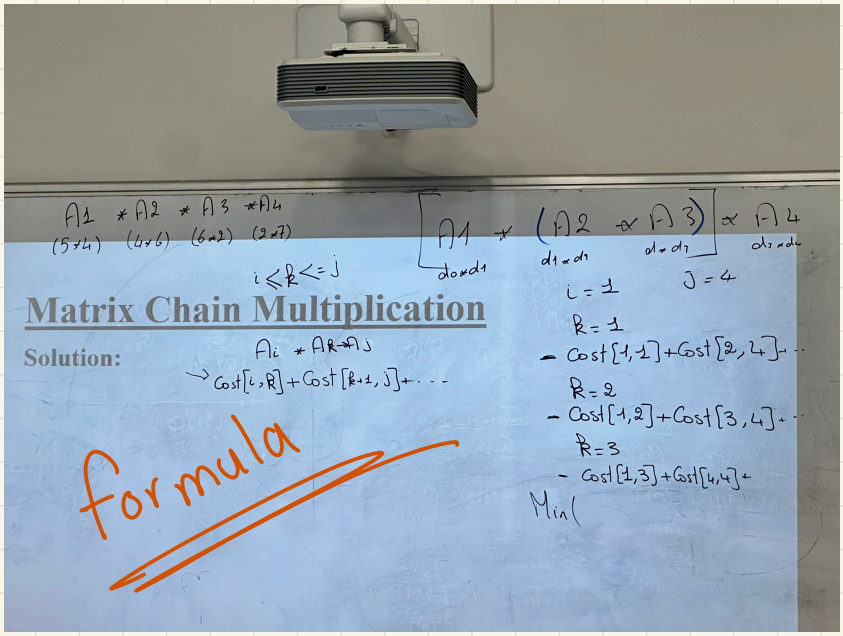


steps to solve Dynamic programming :-

slides

Matrix Chain multiplication :-

- same concept of greedy algorithm.
- see locally what's the best solution.



Non divisible

knapsack

→ in the table I do combinations between 1, 2 and 2, 3 and finally them both (in the 3rd iteam)

and so on for all other following, you should consider the cases before