Course On DP

- Intro to DP
- 1D
- 2D
- 3D & mDP
- tree DP
- non trivial DP problems

Pre-requisite

- Recursion
- Backtracking

for loops, arrays/vectors

HashMap

Agenda → Intro to DP → what, why, where

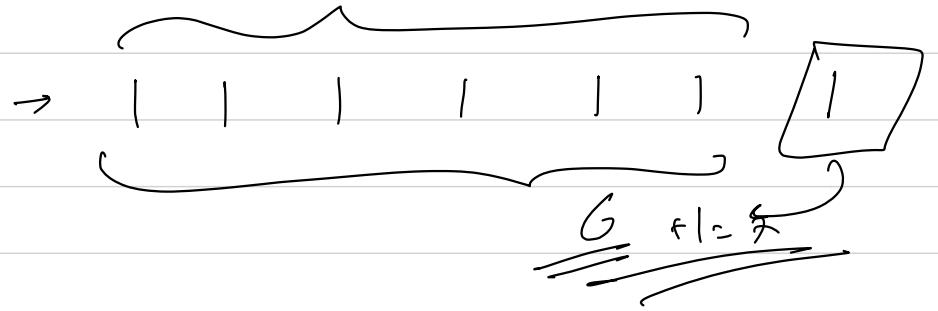
- States, subproblems
- mutual exclusion & mutual exhaustion of recurrences
- types
- problems ←

→ Algorithms

- Brute force
- D N C
- Greedy
- DP

→ Dynamic Programming ←

It is a paradigm of algorithms, that mainly helps to optimise, counting / minimization / maximization problems



repeated computations → optimize using DP.

LinkedIn

→ 0th 1st 2nd 3rd 4th 5th 6th 7th

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

(i-2) (i-1) ith

Given a value n, calculate the nth fibonacci.

→ $f(n) = f(n-1) + f(n-2)$

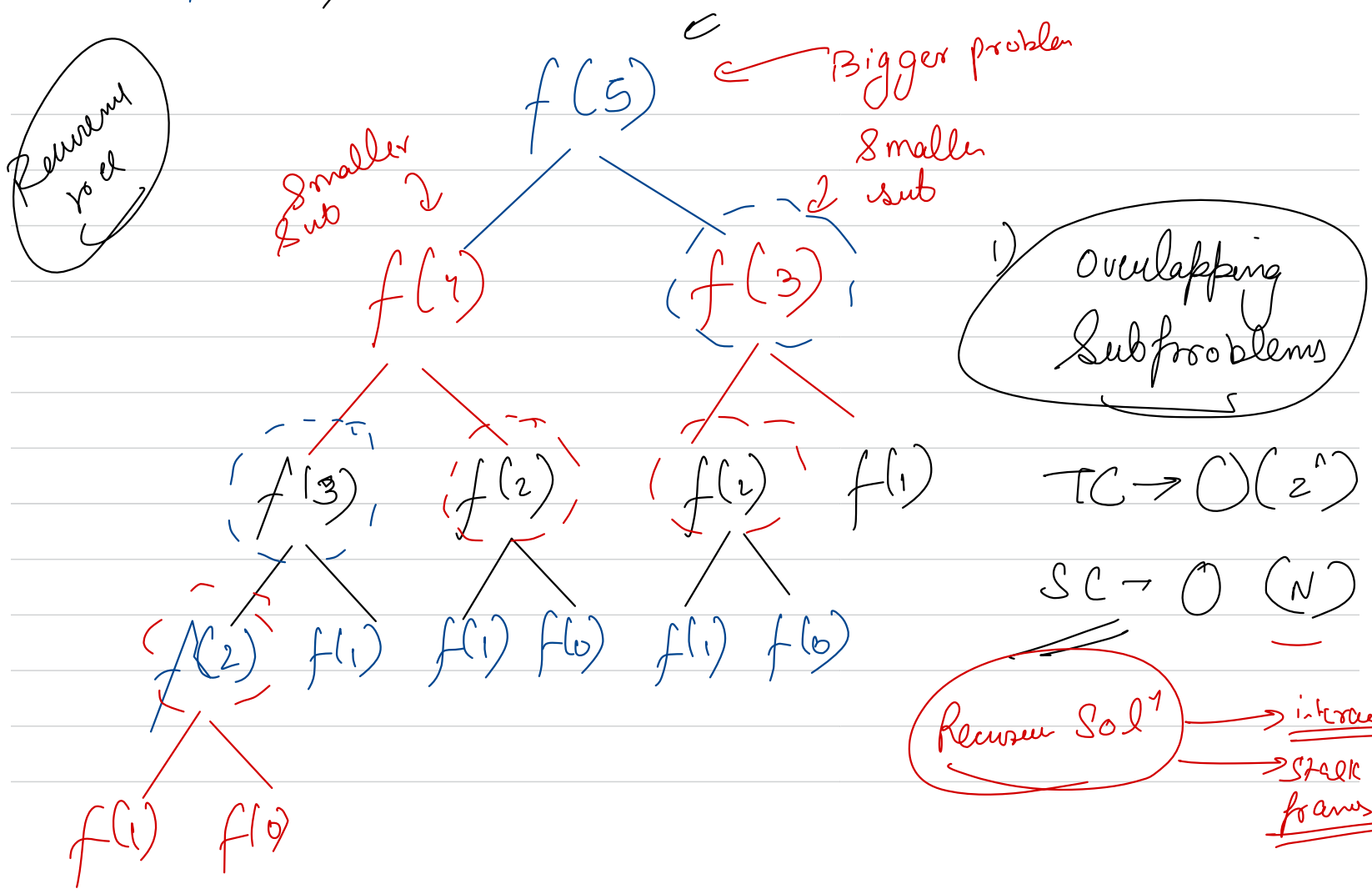
Bigger problem

Smaller & sub problems

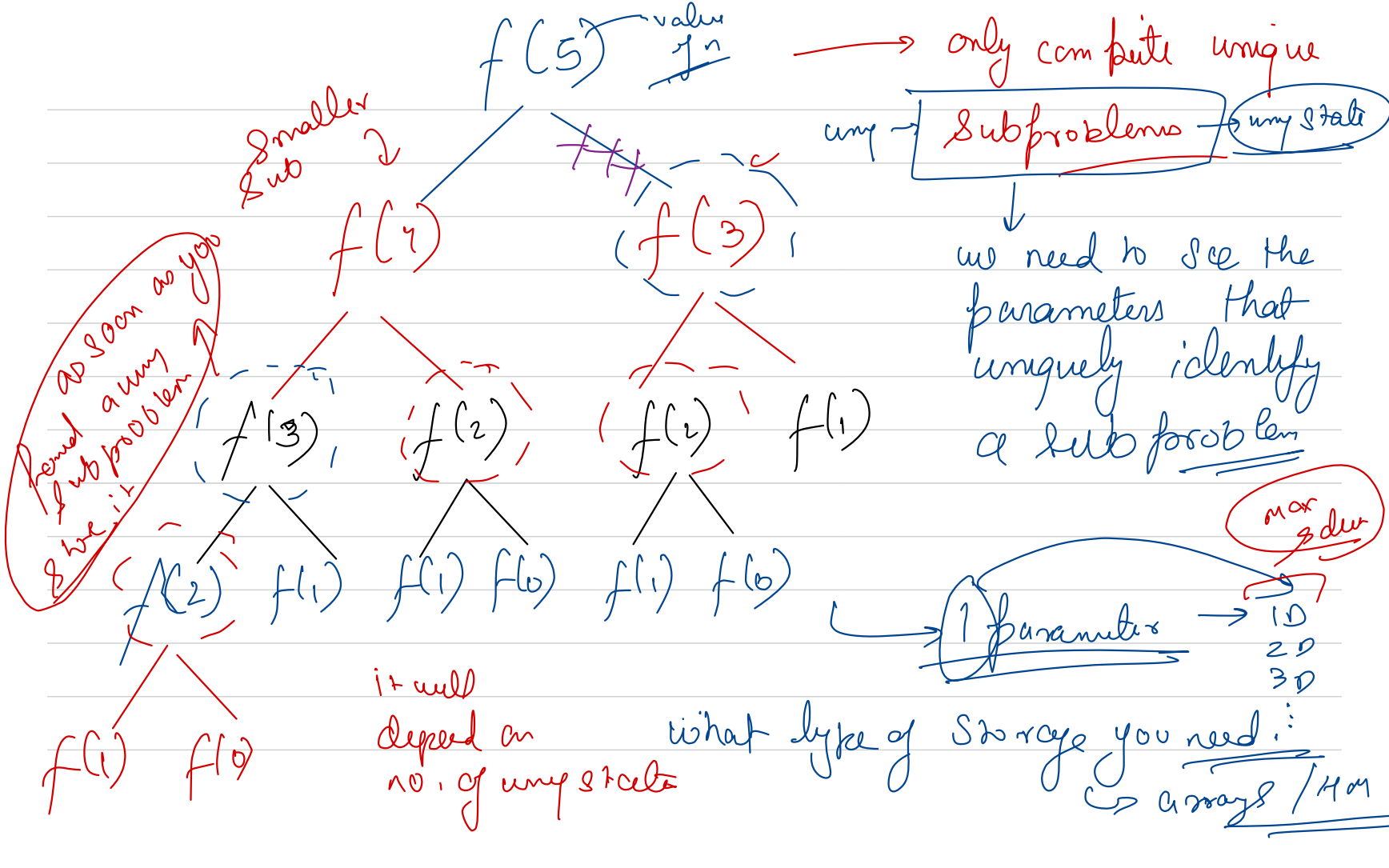
If we assume f → f' that returns the nth fibonacci, correctly

(1) Mutual exclusiveness

(2) Mutual exhaustiveness


Recursive sol
f(5)
Bigger problem
Smaller sub
Smaller sub
f(4)
f(3)
f(3)
f(2)
f(1)
f(1)
f(0)
f(1)
f(0)
f(2)
f(1)
f(1)
f(0)
f(1)
f(0)
1) Overlapping Subproblems
TC → O(2^N)
SC → O(N)
Recursive Sol^n
→ internal DS
→ Stack frames

2) Optimal Substructure → This is a property that states, for a given problem, if you can optimally solve the smaller subproblems, and these smaller subproblems contribute to calculate bigger problem optimally


f(5)
value for n
smaller sub
f(4)
f(3)
f(3)
f(2)
f(2)
f(1)
f(2)
f(1)
f(1)
f(0)
f(1)
f(0)
f(1)
f(0)
Avoid as soon as you find a unq subproblem & store it
only compute unique unq subproblems
unq states
we need to see the parameters that uniquely identify a subproblem
1 parameter
1D
2D
3D
max value
it will depend on no. of unq states
what type of storage you need:
arrays / Hm

dp(i) → gives you i^{th} fib

(dp(i) == -1) → then i^{th} state is yet to be calc.

Two type ⟶ Top Down (Memoization)

Bottom Up (Tabulation)

Base Case

0th 1st 2nd 3rd 4th 5th 6th Fib

0	1	1	2	3	5	8

→ dp array

$dp(i) = dp(i-1) + dp(i-2)$

$dp(i)$ ← i^{th} fib

1D (n+1)

```
c = a + b
a = b
b = c
```

Space optimized:

$dp(i) = dp(i-1) + dp(i-2)$

BU → iterative / tabulation

→ single pass generally

→ can save some space when compared to recursion

TD → recursion / memoize

→ double pass

→ generally here recursive space is same as recursion

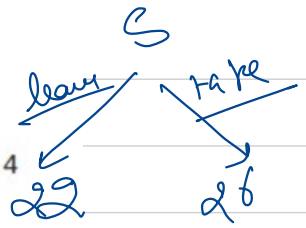
Algo 1 → recursion

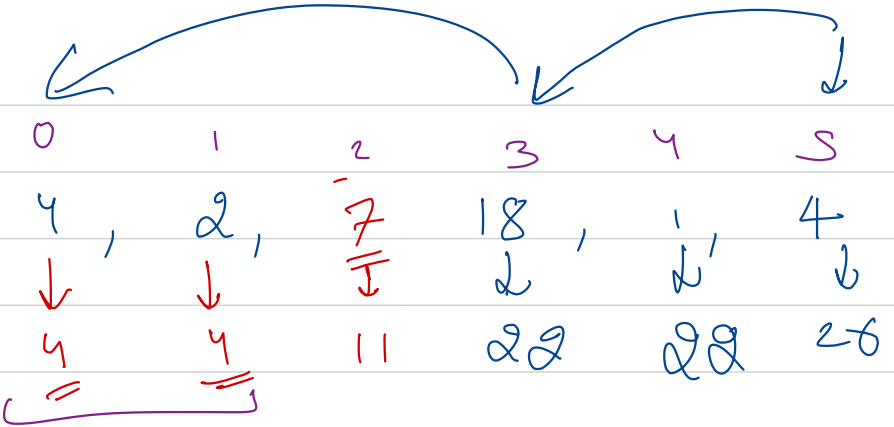
Algo 2 → Top Down DP

House robber

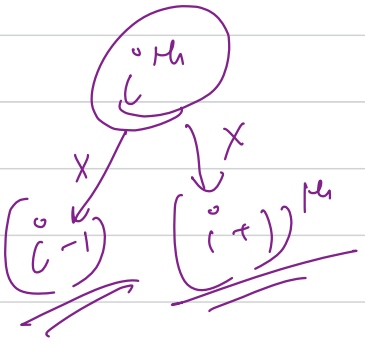
i^{th} → $(i-1)$ $(i+1)$ max profit

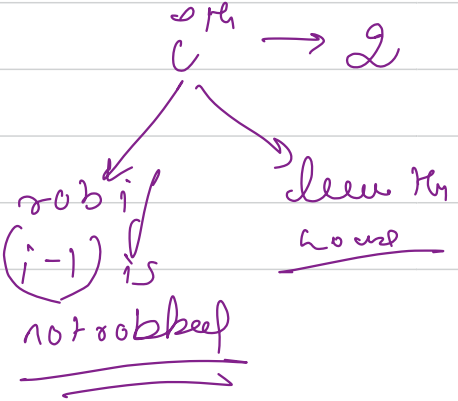
4
7 3 5 1
6
4 2 7 18 1 4
2
9 9
1
8





Base Case





Base Case
Self work
recursion

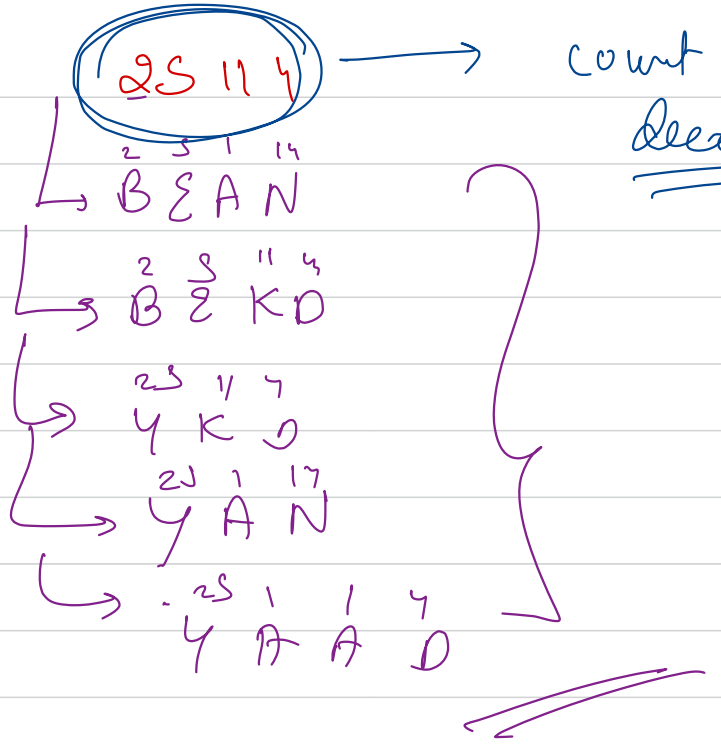
$$f(i) = \max\Big(\underbrace{f(i-1)}_{\text{leave the house}},\ \underbrace{a[i] + f(i-2)}_{\text{rob the house}}\Big)$$

f : that returns max profit till the i^{th} house

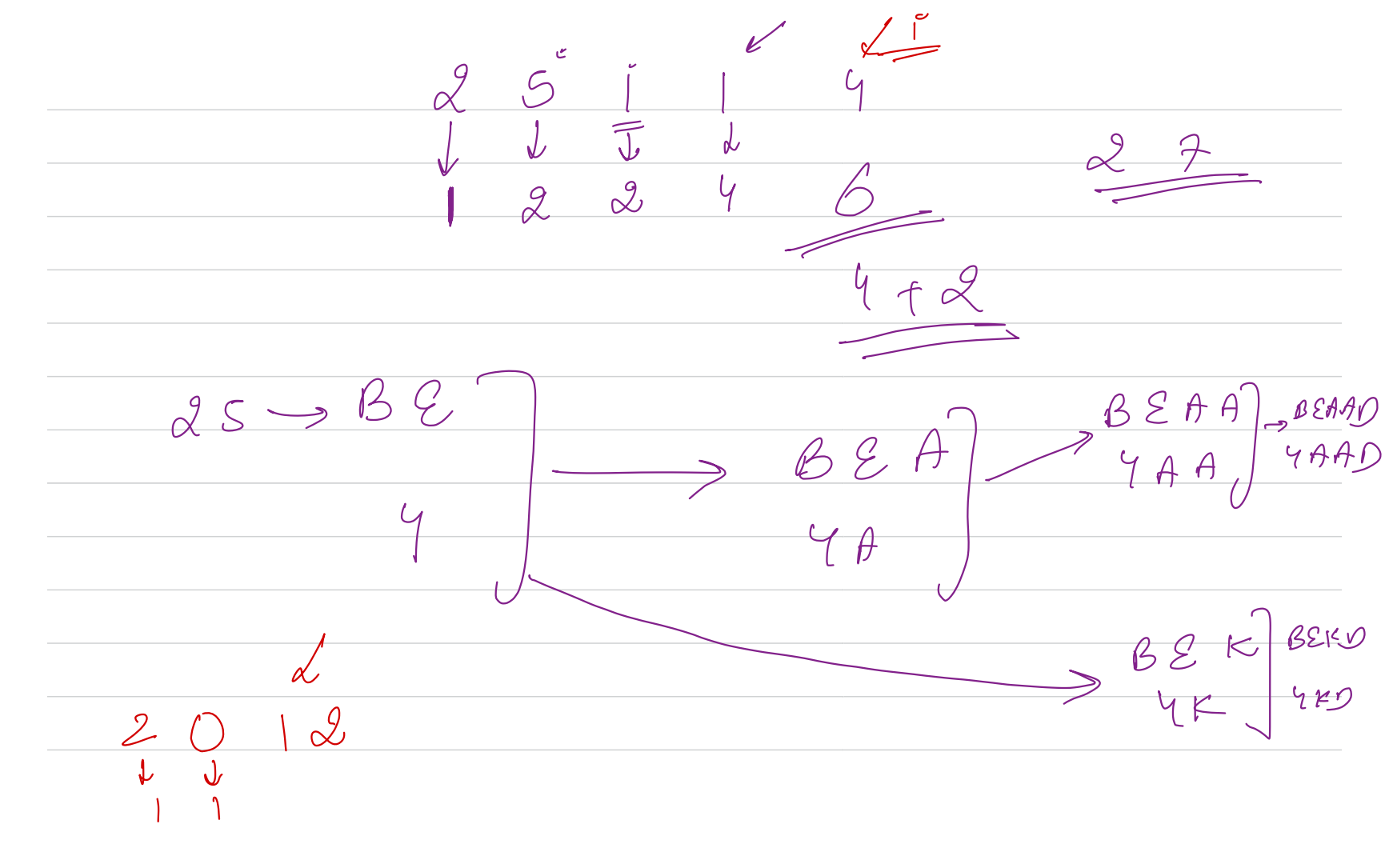
$n \le 10^5$

$$dp(i) = \max(dp(i-1),\ a[i] + dp(i-2))$$

Q Acode Spoj →



count the total decodes


2 5 1 1 4
1 2 2 4
6
2 7
4 + 2
25 → BE
4
BEA
4A
BEAA
4AA
BEAAD
4AAD
BEK
4K
BEKD
4KD
20 12
1 1

$f(i)$ = total no. of ways to decode till the i^{th} index

$$f(i) = \begin{cases} f(i-1) & \text{if } (a[i-1] > 0) \\ + & \\ f(i-2) & \text{if } (a[i-2] > 0 \text{ and } a[i-2]*10 + a[i-1] <= 26) \end{cases}$$

0x → (3) → (10)