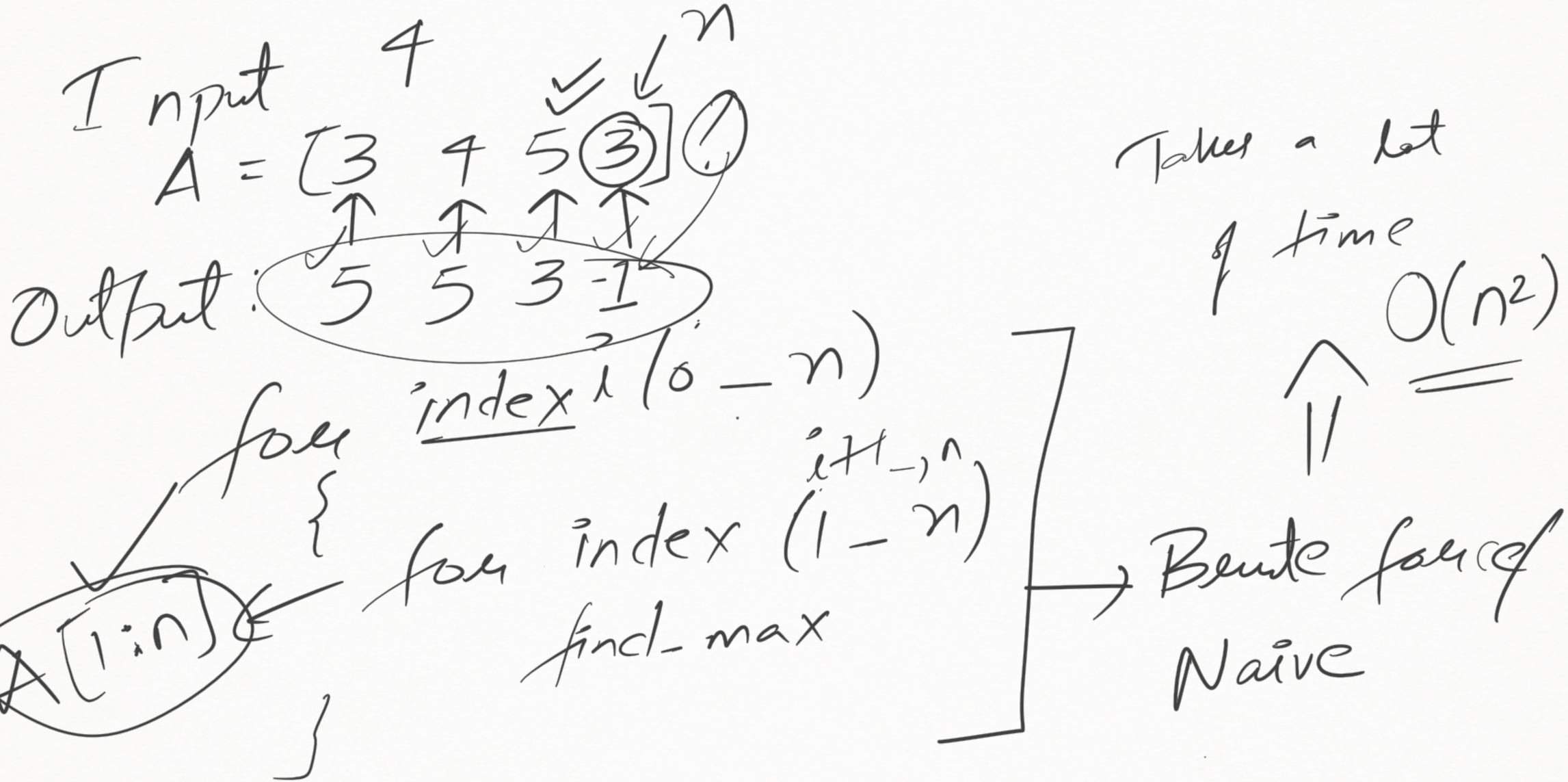
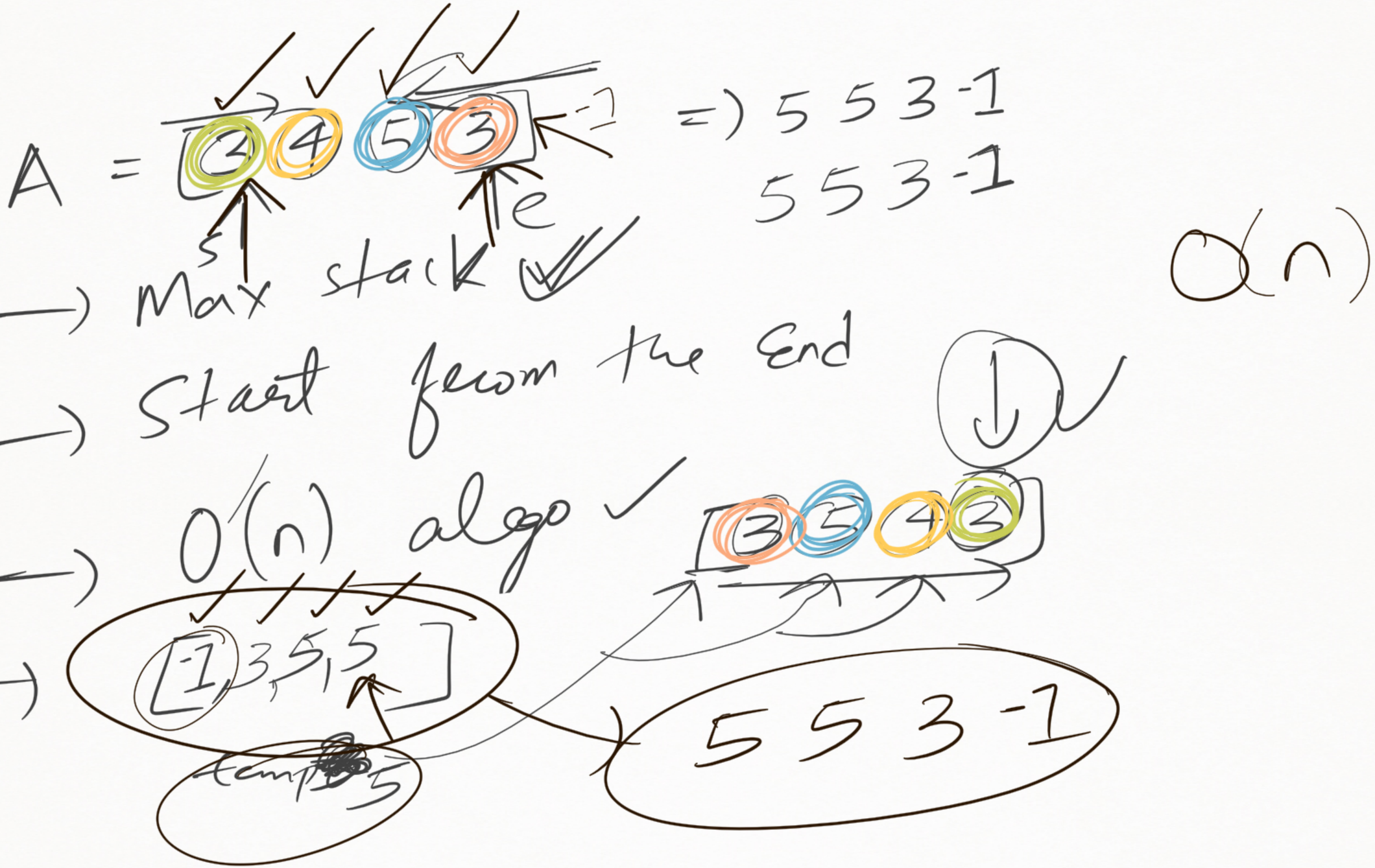

Input
4
A = [3 4 5 3 1]
n
Output: 5 5 3 -1
for 'index' i (0 - n)
{
A[1:n]
for index (i+1 - n)
find_max
}
Takes a lot of time
O(n²)
Brute force / Naive


A = 3 4 5 3 -1
=) 5 5 3 -1
5 5 3 -1
O(n)
s
e
Max stack
Start from the End
O(n) algo
[-1, 3, 5, 5]
3 5 4 3
5 5 3 -1

Revisit Recursion

Problem: Sum all values from 1 to n.

sum_natural(n) ✓

{ }

Solution Step 1> Keep in mind what fun should do?

1 + 2 + _ _ _ n

(R) Pick a sub-prob & assume that your fn already solves it

sum_natural(n-1) - - sum_n(n-2)

- - - sum_natural(1)

Choose the closest subproblem to the original prob

=> (n-1)

```
sum_natural(n)
  {
    sub_sol = sum_natural(n-1)
  }
```

Step 3) Take sub_sol & use it to solve orig prob.

Orig = 1 to n ✓

Sub-sol = 1 to n-1 ✓ + n

```
sum_natural(n)
  { sub_sol = sum_natural(n-1)
    return n + sub_sol
  }
```

Step 4:) Base Case

→ (termination condition)

→ Ending condition

→ Usually it is if else in the beginning.

→ Easiest case, that can be written like Hardcode

$n \rightarrow n-1 \rightarrow n-2 \; \cdots \leftarrow 1$

if n == 1

return 1

```
sum_natural(n)
{   if n == 1
        return 1
    sub_sol = sum_natural(n-1)
    return n + sub_sol
}
```

Example ② Reverse a string Using Recursion

① Input : qwerty

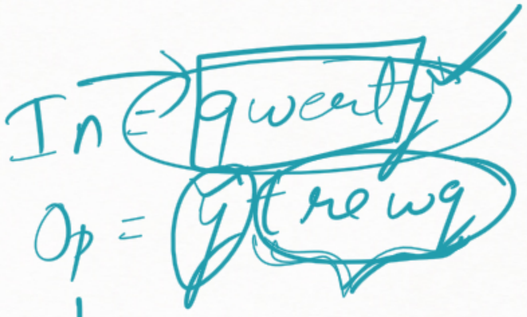
Output : ytrewq

```
reverse_string(input_str)
{



}
```

Step 2) Sub-prob?



String without last char

```
reverse_string(inp_str)
{
    sub-sol = reverse_string(inp_str
                             [0 -> end-1])

}
```

Step 3

sub_sol = inp_str(0→end-1)

I/p = qweert

O/p = treewq

Orig

I/p = qwerty

O/p = ytrewq

```
reverse_str(inp_str)
{
    sub_sol = reverse_str(inp_str[0→end-1])
    return inp_str(end) + sub_sol
}
```

Step 4) Base Case

I/p = qwerty
O/p = ytrewq

qwerty

y + (qwert)
t + (qwer)
r + (qwe)
e + (qw)
w + (q)
q +

(0 → 0)
(0 → 0)

0 → end-1)

if len(inp_str)==1
return inp_str

```
reverse_string(inp_str)
{
    if len(inp_str) == 1
        return inp_str.

    sub_sol = reverse_string(inp_str[0 -> end-1])

    return inp_str(end) + sub_sol

}
```

Common Subsequence

Example ★

string 1 = a b c d e f g h i j

string 2 = c e g j

Both to be in same order

Chars Com Sub have

c, e ✓

c e g j ✓

e g j ✓

c e g ✓

If I say e g j is a common subseq. then e g j should appear in the same order in both strings.

string1 = apoplxlme

string2 = apple

Common subseq → apple, a, app, le

Largest common Subseq.

out of all

string1 = apoplxlme

string2 = apple

ap ple

LCS

aple

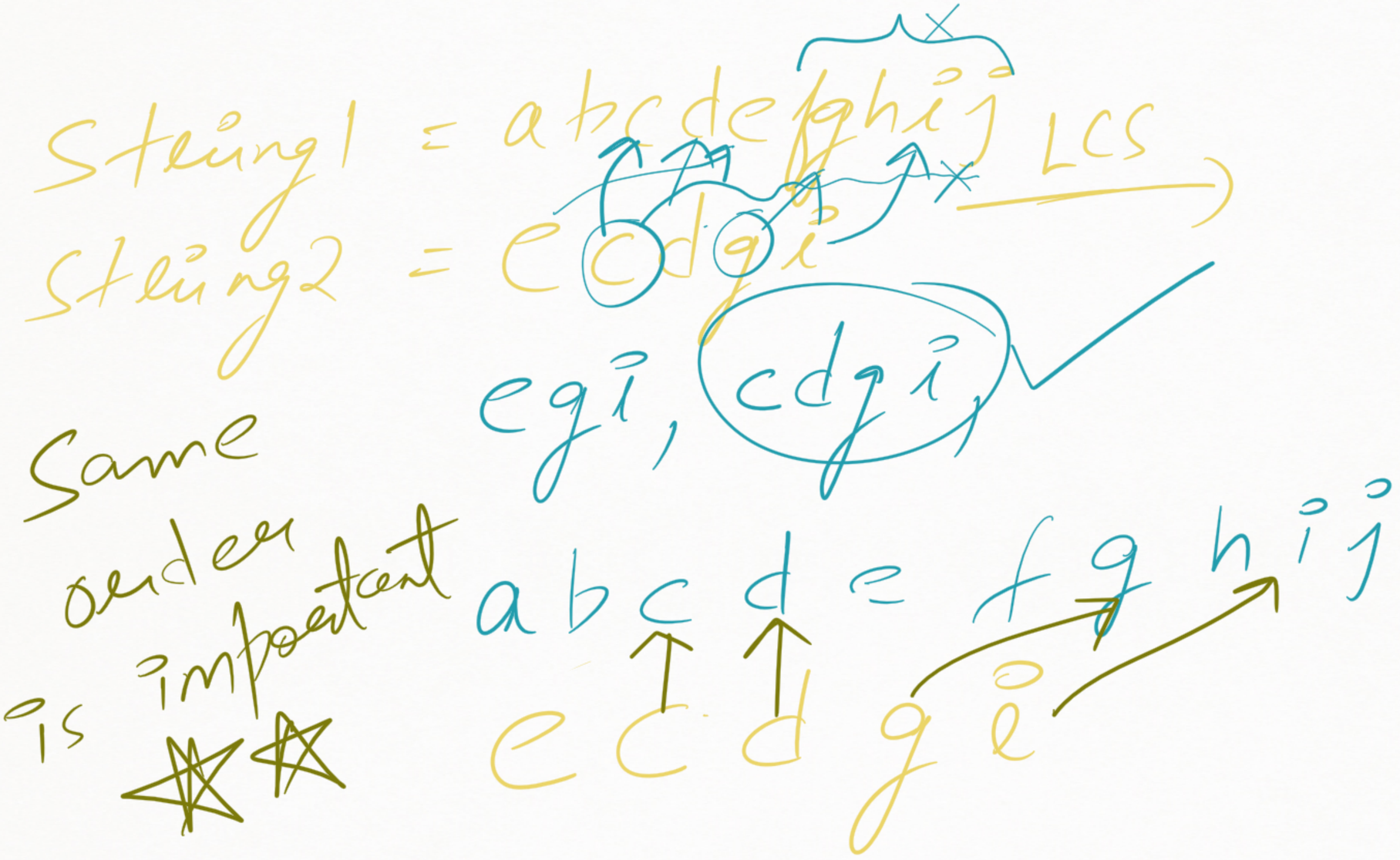
A = apoplxlpme

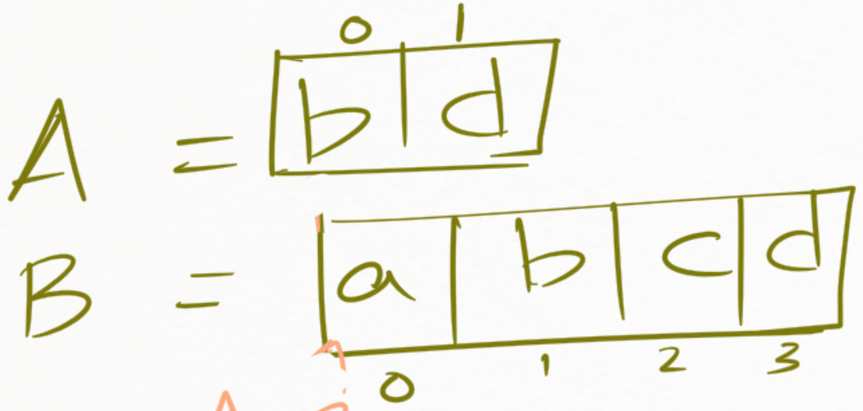
B = apple

LCS ⇒ apple, aple?

4

o/p : length of LCS


string1 = abcdefghij
string2 = ecdgi
LCS
egi, cdgi
Same order is important
abcd = fghij
ecdgi



Brute force/Naive?

A = b d e

B = a x d e

i

Step 1

A, B

LCS ($\vec{i}$, $\vec{j}$)

{

}

Input => Pointers to both strings

O/p => length of LCS

return => length of LCS

Step 2

★ Recursion with optimisation → Longest
Decision

Pick a Sub-prob ⇒ Reduced Size of Inp-str

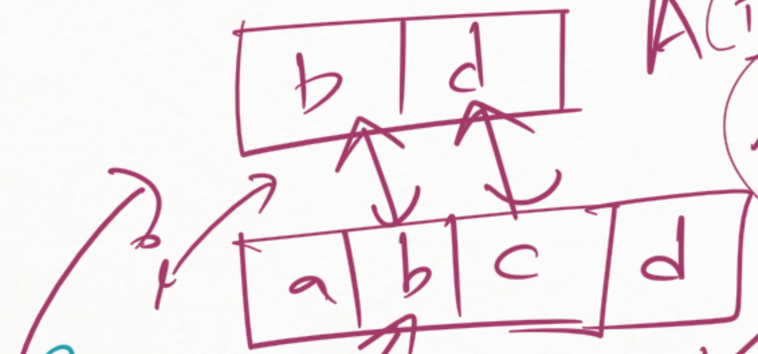
A = b d

B = a b c d

i, j

Match?

Non Match?



A[i] == A[j] take the diag

i → i+1

j → j+1

Step 3

LCS(i+1, j+1) + 1

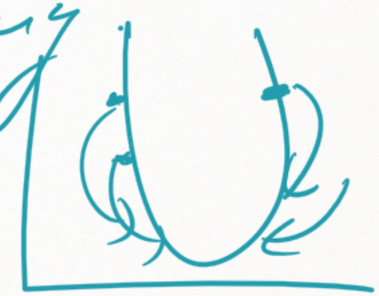
A[i] ≠ A[j]

Step 3

max {

LCS(i, j+1)

LCS(i+1, j)



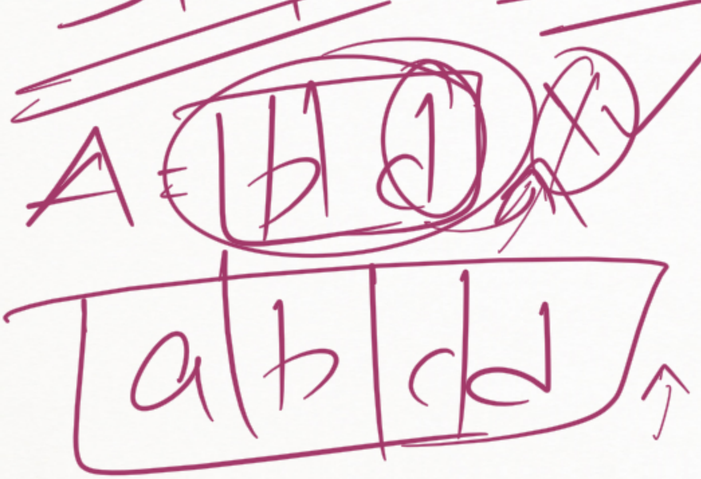
Step 2 continue

```
LCS(i, j)
{
  if (A[i] == A[j])
      sub_sol = LCS(i+1, j+1)
  else
      sub_sol-1 = LCS(i+1, j)
      sub_sol-2 = LCS(i, j+1)
}
```

Step 3


```
LCS(i, j)
{
    if A[i] == B[i]
        sub_sol = LCS(i+1, j+1)
        return 1 + sub_sol
    else
        sub_sol_1 = LCS(i+1, j)
        sub_sol_2 = LCS(i, j+1)
        return max(sub_sol_1, sub_sol_2)
}
```

step 4 Base Case

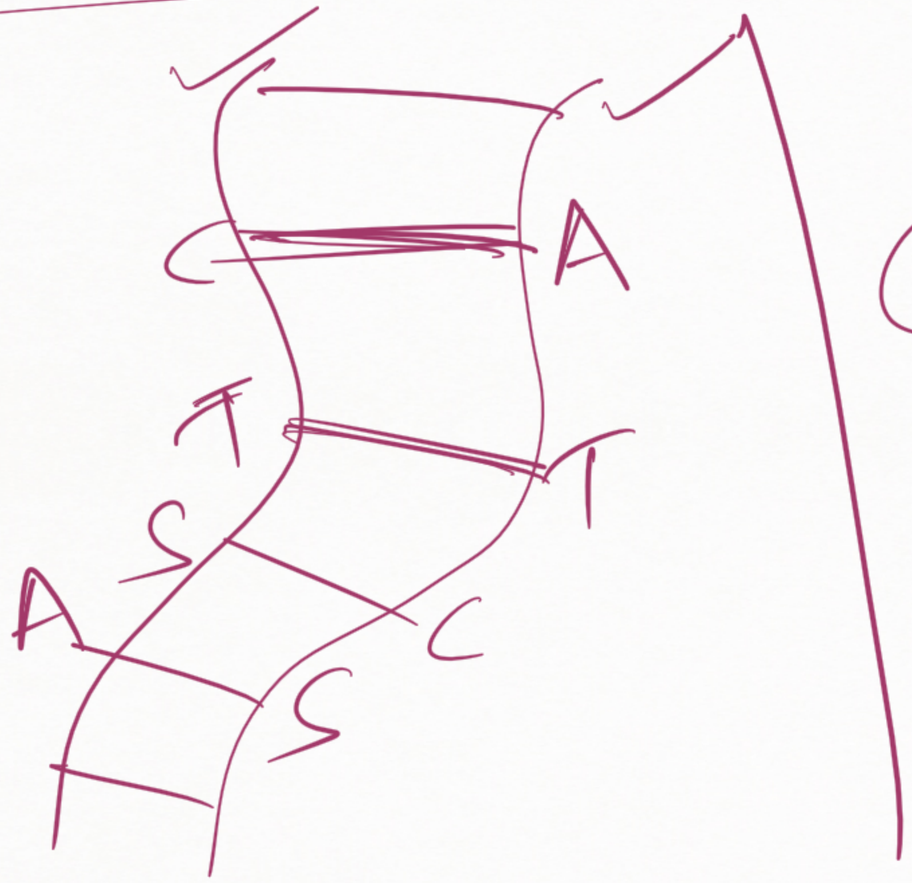


if anyone of them becomes empty/end

```
if i == len(A) or
   j == len(B)
return 0
```

```
LCS(i, j)
{
    if i == len(A) or
       j == len(B) : return 0
    else if A[i] == B[i]
        return 1 + LCS(i+1, j+1)
    else
        return max(LCS(i+1, j), LCS(i, j+1))
}
```

DNA Strands



App of LCS

Edit Distance

- Integrity of Master Data Management Solutions