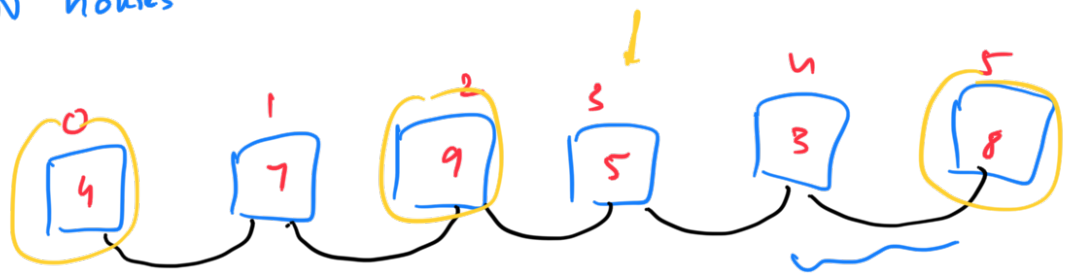
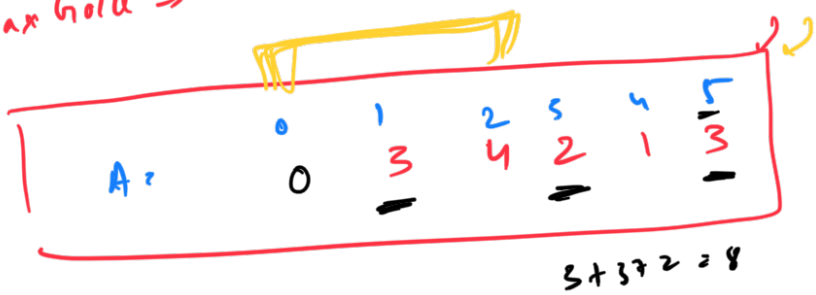
DP - 2

Question: Robbery

N houses which are connected by alarms



Max Gold ⇒ 4 + 9 + 8 = 21



maxGold(2)

maxGold(5)

⇒ 1 + 4 + 3

3 + 3 + 2 = 8

Assumption: 2

maxGold(i) ⇒ Returns the maximum gold that I can steal till ith house

A = 10 3 6 (0 1 2)

maxGold(1)

Ans -

maxGold(5)

maxGold(n-1)

gold > 0

A = 4 7 9 5 3 8 (0 1 2 3 4 5)

maxGold(0) = A[0];

maxGold(1) = max(A[0], A[1]).

(i-2)

dp[N]

choices :

1) Steal from the ith house

goldi = A[i] + maxGold(i-2) ✓

2) Don't steal

gold2 = maxGold (i-1)

maxGold (i) = max (A[i] + maxGold (i-2), maxGold (i-1))

Overlapping Subproblems : YES

Dynamic Programming

T.C : O(# states × T.C Per state)

N × O(1)

T.C : O(N)

S.C : O(N) → O(1)

TOP-Down → Bottom-Up

Question : Longest Increasing subsequence

Ans = 4

A = 0 1 2 3 4 5 6
1 3 2 3 1 2 5

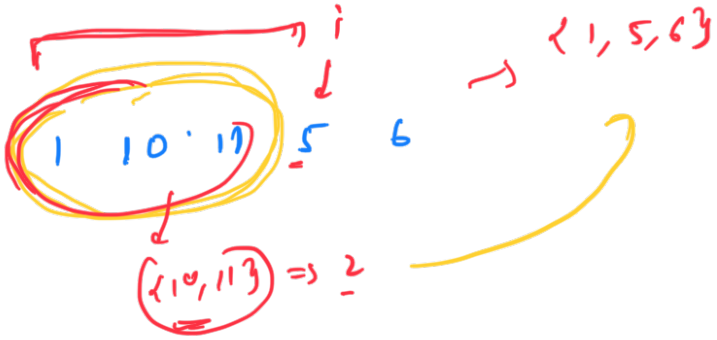
X

2 1 3

1 3 5 → 3

1 3 3 5

1 2 3 5 → 4



Assumption:

LIS(i) => Length of Longest increasing subsequence ending at index 'i'

A =

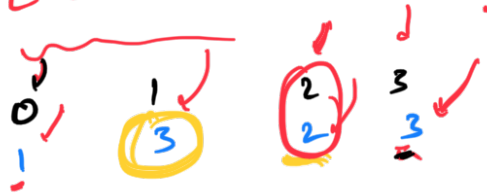
0	1	2	3	4	5	6
1	3	2	3	1	2	5

LIS(3) = {1, 2, 3} => 3

LIS(4) = {1} => 1

LIS(2) = {1, 2} => 2

LIS(6) = {1, 2, 3, 5} => 4

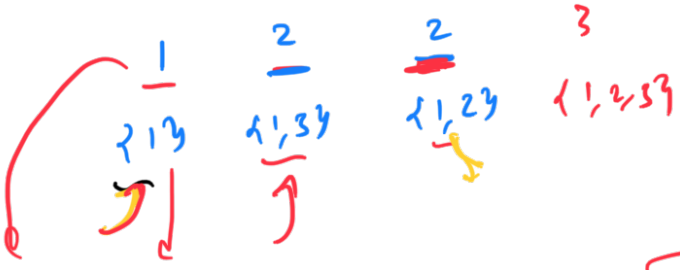


LIS(3) ?

5 > 2

A =

LIS()



{1, 2, 3} => 3

{1, 3} => 2

LIS(

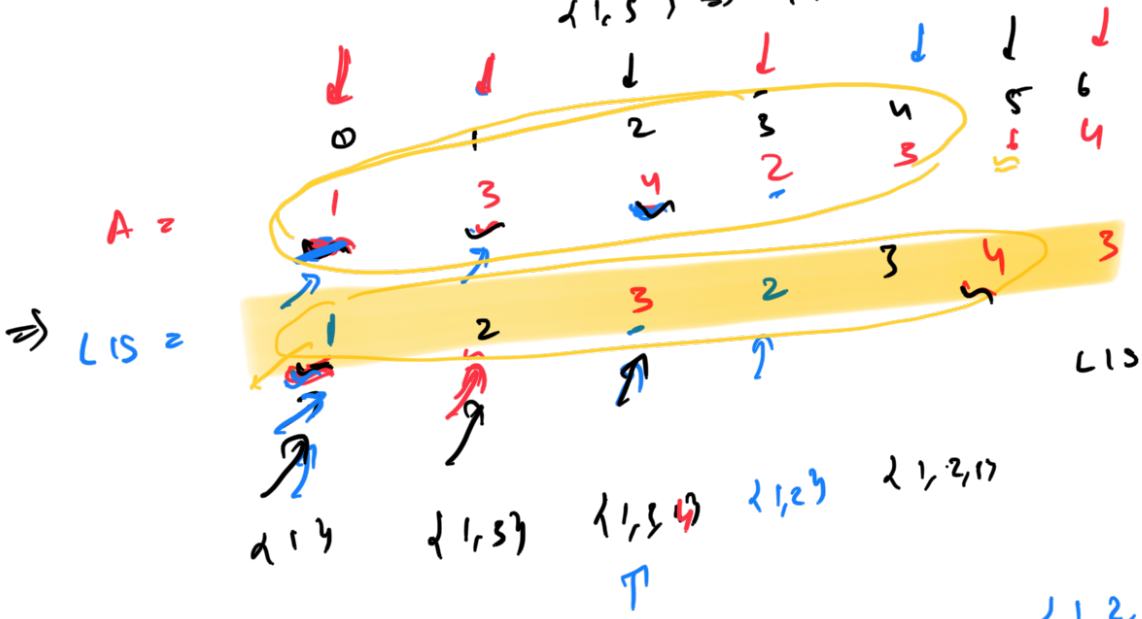
LIS(3)

LIS(6)

{1,2,5} => 3
{1,5} => 2
{1,2,3,5} => 3+1=4
{1,2,5} => 2+1=3
{1,3,5} => 2+1=3
{1,5} => 1+1=2

{1,2} => 2
{1,2} => 2

Max (i)
A[j] < A[i]



{1,2,3} => 3
{1,3} => 2

Overlapping subproblems

LIS(5) => LIS(4), LIS(3), LIS(2), LIS(1), LIS(0)

LIS(0) => 5, 4, 3 ... 2

dp[] = {-1};

Bottom-UP

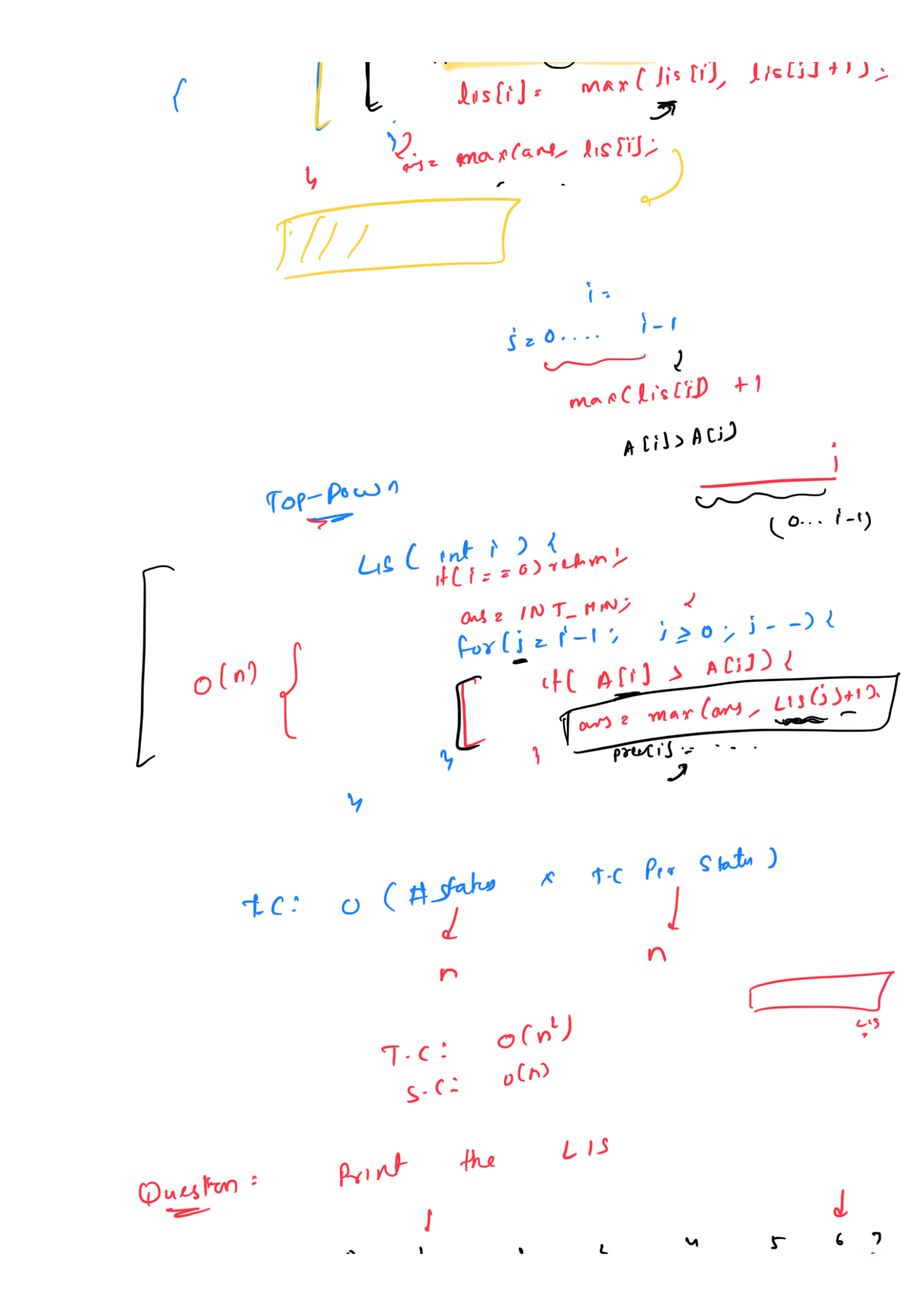
LIS[N] = {-1};

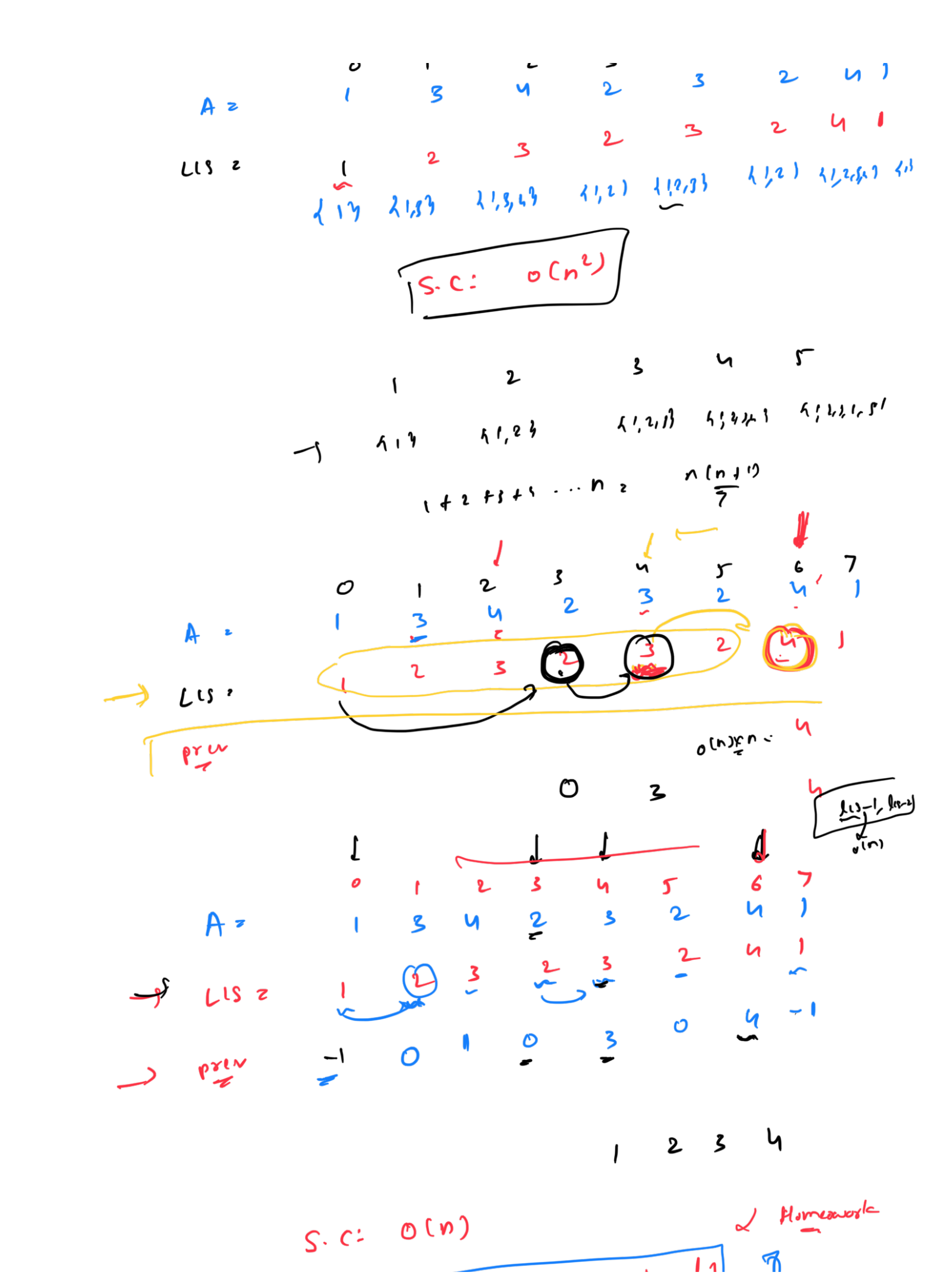
LIS[0] = 1

LIS(i)

(j <= i-1) => j < i

for (i = 1; i < n; i++) { -> LIS[i]
  lis[i] = 1;
  for (j = 0; j <= i-1 ; j++) {
    if (A[i] > A[j])


lis[i] = max(lis[i], lis[j] + 1);
ans = max(ans, lis[i]);
i =
j = 0 .... i-1
max(lis[j]) + 1
A[i] > A[j]
(0 ... i-1)
Top-Down
Lis(int i) {
if(i == 0) return 1;
ans = INT_MIN;
for(j = i-1; j >= 0; j--) {
if(A[i] > A[j]) {
ans = max(ans, Lis(j)+1);
prevIs = ...
O(n)
T.C: O(# states * T.C Per State)
n
n
Lis
T.C: O(n^2)
S.C: O(n)
Question: Print the LIS
1
4
5
6
7


A =
LIS =
S.C: O(n²)
LIS =
prev
O(n)
A =
LIS =
prev
Homework

~~Binary~~ → Binary Search [Upper bound]

Segment Tree :

Question : Longest Common Subsequence

S1 = a b b c d f

S2 = b a b d c f

abcd, abdf, bbdf

Ans = 4

S1 = a g g t a t

S2 = g x t x a y b

} gta = 3 3

S1 = abcd

S2 = dcba

} LCS(S1, S2) = 1

Recursion ?

Assumption :

LCS(l_1, l_2) = Length of longest Common Subsq of S_1 upto l_1 and S_2 upto l_2

Case1 : Last characters of S_1 and S_2 are equal

S_1 = a b b c d f

S_2 = b a b d c f

l_1 = 6

l_2 = 6

$LCS(l_1, l_2)$ = 1 + $LCS(l_1-1, l_2-1)$;

Cases: Last characters are not equal

$LCS(S_1, S_2)$ = ka

a s c ✗

a ✓ c

Option 1:

S_1 = l k a → $LCS(l_1, l_2-1)$;

S_2 = k b a

Option 2:

S_1 = l k $LCS(l_1-1, l_2)$

S_2 = k b a c

$LCS(l_1, l_2)$

if($S_1[l_1-1] == S_2[l_2-1]$) {

$LCS(l_1, l_2)$ = 1 + $LCS(l_1-1, l_2-1)$;

}

else {

$LCS(l_1, l_2)$ = max($LCS(l_1-1, l_2)$, $LCS(l_1, l_2-1)$;

Top-down

dp[l1][l2] = ?

if(l1 == 0)

LCS(l1, l2) = 0

if(l2 == 0)

LCS(l1, l2) = 0

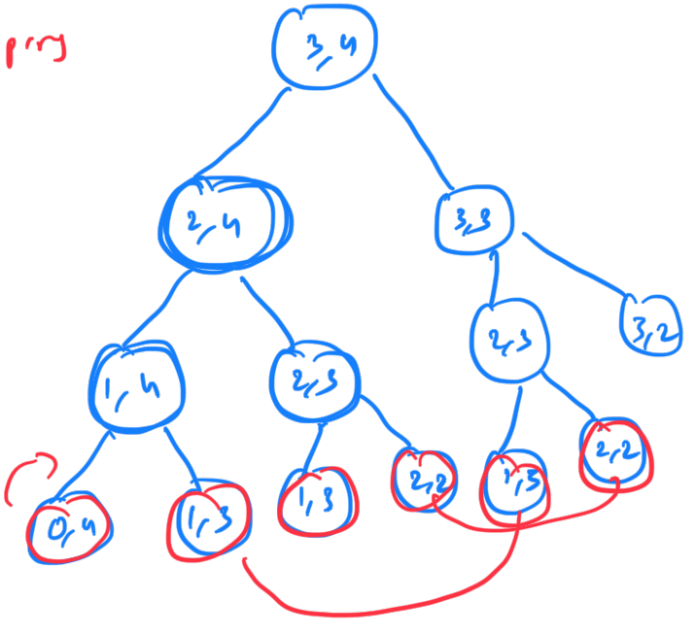
S1 = ?

S2 = "abcdf"

S1 = 0 1 2
l k a

S2 = k b a c
0 1 2 3

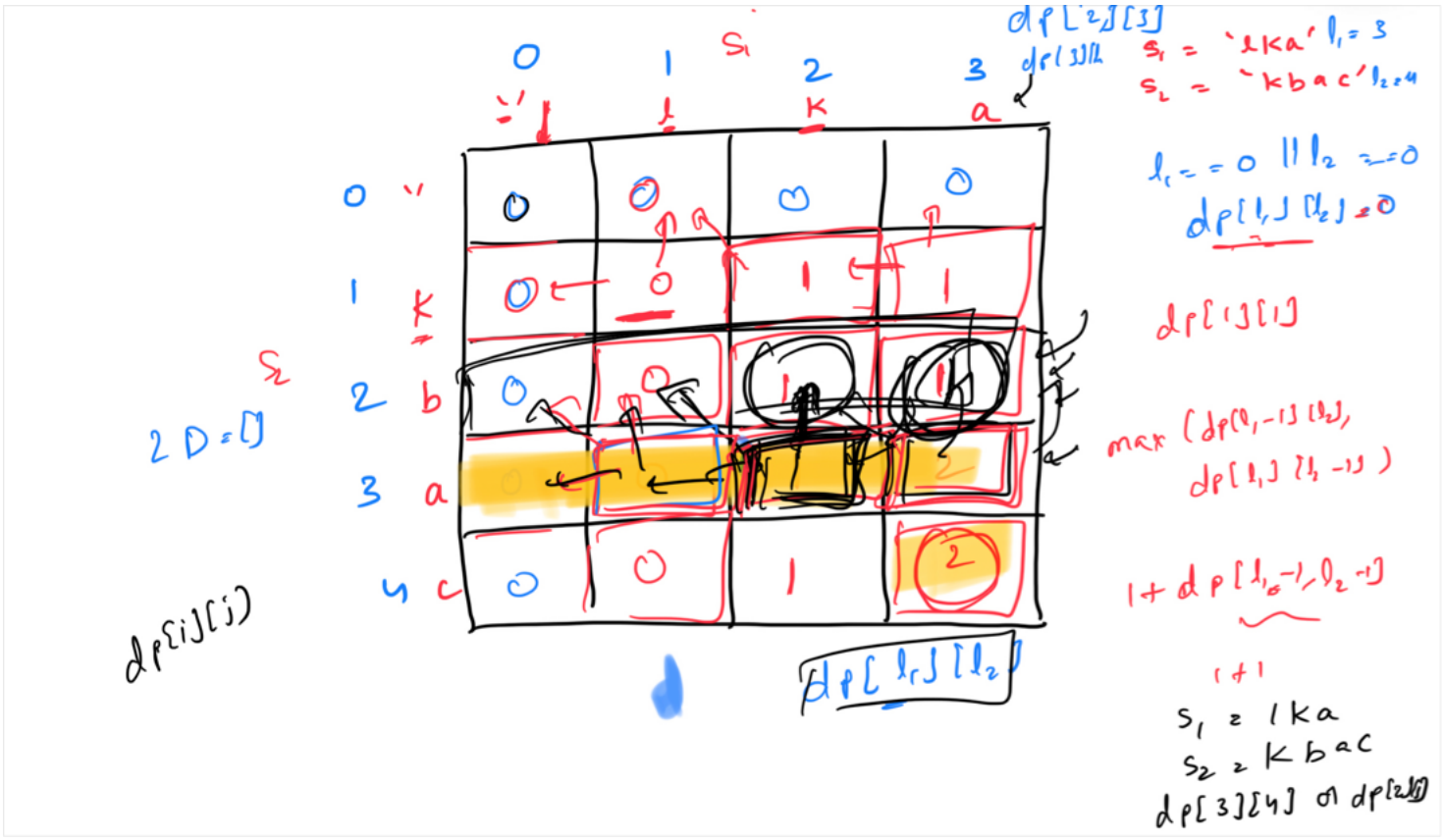
Overlapping



dp → 1D array

TOP- DOWN →

Bottom - Up


dp[2][3]
dp[3][4]
S1
S2
2D dp[][]
dp[i][j]
dp[l1][l2]
s1 = 'lka' l1 = 3
s2 = 'kbac' l2 = 4
l1 == 0 || l2 == 0
dp[l1][l2] = 0
dp[i][j]
max (dp[l1-1][l2], dp[l1][l2-1])
1 + dp[l1-1][l2-1]
i+1
s1 = lka
s2 = kbac
dp[3][4] or dp[2][3]

```
dp[l1+1][l2+1];
for(i=0; i<l1; i++)
    dp[i][0] = 0;
for(i=0; i<l2; i++){
    dp[0][i] = 0;
}
for(i=1; i<=l1; i++){
    for(j=1; j<=l2; j++){
        if(s1[i-1] == s2[j-1])
            dp[i][j] = 1 + dp[i-1][j-1];
        else
            dp[i][j] = max(dp[i][j-1],
                           dp[i-1][j]);
    }
}
```

O(1) [ ]

T.C: O(# states × T.C per state)

states = $l_1 \times l_2$

T.C per state = O(1)

= $O(l_1 \times l_2)$

S.C: $O(l_1 \times l_2) \rightarrow O(l_1)$

$O(l_1)$ ← $O(\min(l_1, l_2))$

Bottom Up

Question: Edit Distance

s_1 = sunday

s_2 = saturday

Convert s_1 to s_2 using minimum no. of operations

1) Insert in S_1
2) Replace a char in S_1
3) Delete a character in S_1

S_1 = S u n d a y → Saturday

S_2 = S a t u r d a y

Strategy 1:

Rep(u, a) → Rep(n, t) ; Insert(r), Insert(z)

u operations

Strategy 2: Insert(a), Insert(t), Rep(n, r)

3 operations

S_1 = T u e s d a y

S_2 = T h u r s d a y

2 operations

S_1 = c d a b

S_2 = c d a

len(S_2) − LCS(S_1, S_2)

3 − 1 = 2

3 operations

minCost(l_1, l_2) ⇒ Gives the minimum no of operations to convert S_1 upto l_1 and S_2 upto l_2

Case 1: Last characters are equal

(S u n d a y)

S_1 = s u n d a y

S_2 = s a t u r d a y

minCost(l_1, l_2) = minCost($l_1 - 1$, $l_2 - 1$);

Cases: Last characters are not equal

S_1 = s u n

S_2 = s a t u r

Option1: Insert

s u n r

s a t u r

1 + minCost(l_1, $l_2 - 1$)

Option2: Replace

S_1 = s u r

S_2 = s a t u r

1 + minCost($l_1 - 1$, $l_2 - 1$)

Option3: Delete

S_1 = s u r

S_2 = s a t u r

1 + minCost($l_1 - 1$, l_2);

```
int minCost(l1, l2) {
    if(l1 == 0) return l2;
    if(l2 == 0) return l1;
    if(dp[l1][l2] != -1) return dp[l1][l2];
    if(s1[l1 - 1] == s2[l2 - 1]) {
        dp[l1][l2] = minCost(l1 - 1, l2 - 1);
```

return dp[l1][l2]

} else {

dp[l1][l2] = 1 + min (minCost(l1-1, l2)
minCost(l1-1, l2-1)
minCost(l1, l2-1));

return dp[l1][l2]

}

}

S1 = "" → l2

S2 = "abcde"

S1 = "abcd" → l1

S2 = "e"

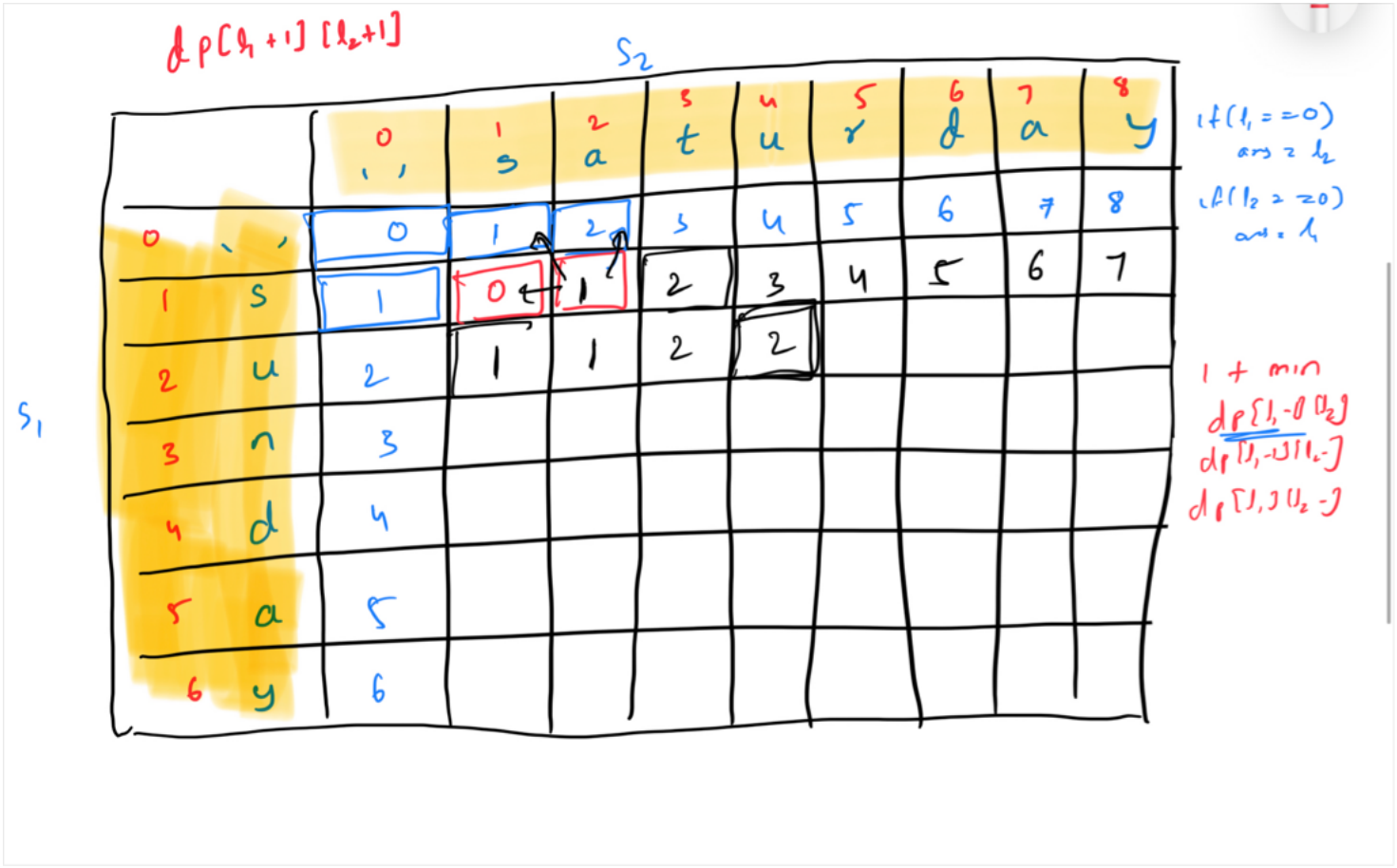
T.C: O(# states * T.C per state)

l1 * l2 * O(1)

T.C: O(l1 * l2)

S.C: O(l1 * l2)

Bottom-Up

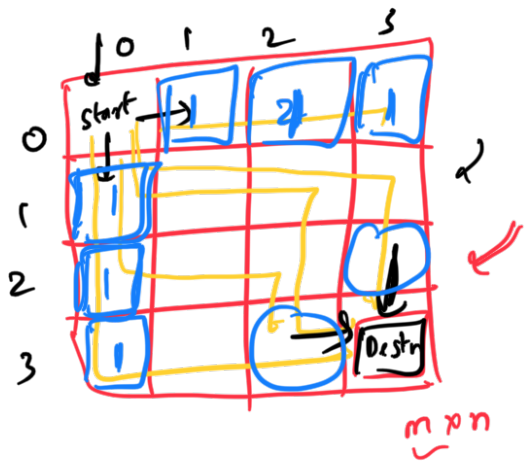

dp[l1+1][l2+1]
S2
S1
0 ' '
1 s
2 a
3 t
4 u
5 r
6 d
7 a
8 y
0 ' '
1 s
2 u
3 n
4 d
5 a
6 y
if(l1 == 0) ans = l2
if(l2 == 0) ans = l1
1 + min
dp[l1-1][l2]
dp[l1,-1][l2-]
dp[l1,][l2-]

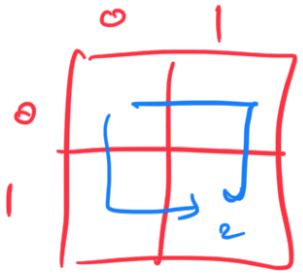
```
// fill 1st row & 1st coun

for (i=1; i<=l1; i++){

    for(j=1; j<=l2; j++) {
        if( s1[i-1] == s2[j-1] {
            dp[i][j] = dp[i-1][j-1];
        }
        else
            1 + min(dp[i-1][j], dp[i-1][j]
                                dp[2][j-1]
    }
}
```

T.C: $O(l_1 \times l_2)$

S.C: $O(l_1 \times l_2) \rightarrow O(l_1)$

Question: ways to Reach





Approach1 : Stairs

ways(i,j) ⇒ No. of ways to reach the cell (i,j) from (0,0)

ways(i,j) = ways(i-1,j) + ways(i,j-1)

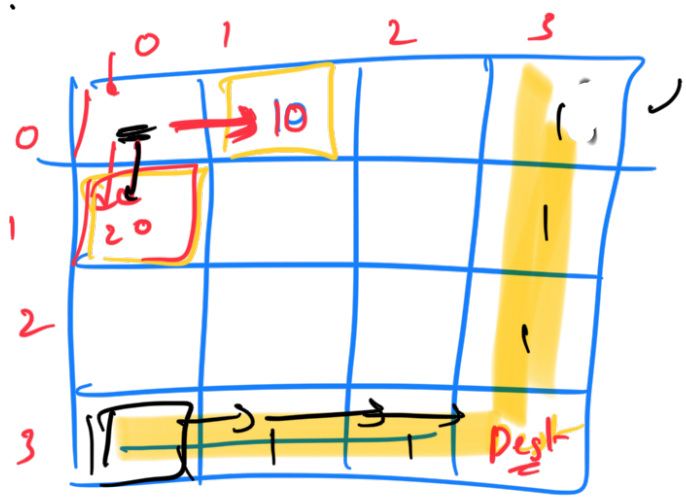
[ways(i,j) = ways(i-1,j) + ways(i,j-1)

if(i == 0 || j == 0)
    return 1;

T.C: O(m*n) * O(1) = O(m*n)

S.C: O(m*n) → O(m)
↓
Bottom-up

Approach 2 :



ways(i,j) = No. of ways to reach destination (n-1, m-1) from (i,j)

ways(i,j) = ways(i+1,j) + ways(i,j+1)

Base Case

(i == n-1 || j == m-1)
ways(i,j) = 1

Question : Robbery

```
dp[N] = {-1};
int maxGold(int gold[], int N){
      if(dp[n] != -1) return dp[n];
      if(n == 0) return dp[n] = A[0];
      if(n == 1) return dp[n] = max(A[0], A[1]);

      dp[n] = max(gold[N] + maxGold(N-2), maxGold(N-1));
      return dp[n];
}
ans = maxGold(gold, N-1);  // N = gold.length();
```

Longest Increasing Subsequence

```
dp[0] = 1;            // Smaller Problem
for(int i = 1; i < n; i++){
      dp[i] = 1;
      for(int j = i-1; j >= 0; j--){
            if(A[j] < A[i]){
                  dp[i] = max(dp[i], dp[j] + 1);
            }
      }
}
```

Longest Common Subsequence: Top Down

```
dp[la+1][lb+1] = {-1};
int lcs(string A, string B, int la, int lb){
      if(la == 0 || lb == 0) return 0;

      if(dp[la][lb] != -1) return dp[la][lb];

      if(A[la - 1] == B[lb - 1])
            dp[la][lb] = 1 + lcs(A, B, la-1, lb-1);
      else
            dp[la][lb]= max(lcs(A, B, la-1, lb),lcs(A, B, la, lb-1));
      return dp[la][lb];
}
```

Longest Common Subsequence: Bottom-Up

```
int LCS(string A, string B){
      la = A.length();
      lb = B.length();
      int lcs[la+1][lb+1];
      for(int i = 0; i < la; i++) lcs[i][0] = 0;
      for(int i = 0; i < lb; i++) lcs[0][i] = 0;

      for(int i = 1; i <= la; i++){
            for(int j = 1; j <= lb; j++){
                  if(A[i-1] == B[j-1])
                        lcs[i][j] = lcs[i-1][j-1] + 1;
                  else
                        lcs[i][j] = max(lcs[i-1][j], lcs[i][j-1]);
            }
      }
return lcs[la][lb];
}
```

Edit Distance : Top Down

```
edit(A, B, l1, l2){
      if(l1 == 0) return l2;
      if(l2 == 0) return l1;

      if(dp[l1][l2] != -1) return dp[l1][l2];
      if(A[l1 - 1] == B[l2 - 1])
            dp[l1][l2] = edit(A, B, l1-1, l2 - 1);
      else{
            dp[l1][l2] =  1 + min( edit(A, B, l1, l2 - 1),
                        edit(A, B, l1-1, l2 - 1),
                        edit(A, B, l1-1, l2))
      }
      return dp[l1][l2];
}
```

Top-Down : Ways to reach (r-1, c-1) from (0, 0)

```
int ways(int i, int j){
      if(i == 0 || j == 0) return 1;
      if(dp[i][j] != -1) return dp[i][j];

      dp[i][j] = ways(i-1, j) + ways(i, j - 1);
return dp[i][j];
}
ways(r-1, c-1)
```

Bottom-up: Ways to reach (r-1, c-1) from (0, 0)

```
int ways(int r, int c){
      ways[r][c] = {0};
      ways[0][0] = 1;
      for(int i = 0; i < r; i++) ways[i][0] = 1;
      for(int i = 0; i < c; i++) ways[0][i] = 1;

      for(int i = 1; i < r; i++){
            for(int j = 1; j < c; j++){
                  ways[i][j] = ways[i-1][j] + ways[i][j-1];
            }
      }
      return ways[r-1][c-1];
}
```