length = ~~1~~ 2 3 4 ~~5~~

[1,17,5,10,13,15,10,5,16,8]

I → {1,17}
D → {17}

I → {1,5,10,13}
D → {1,13,10}

I → {1,17,5,15}
D → {17,15}

I → {1}
D → {13}

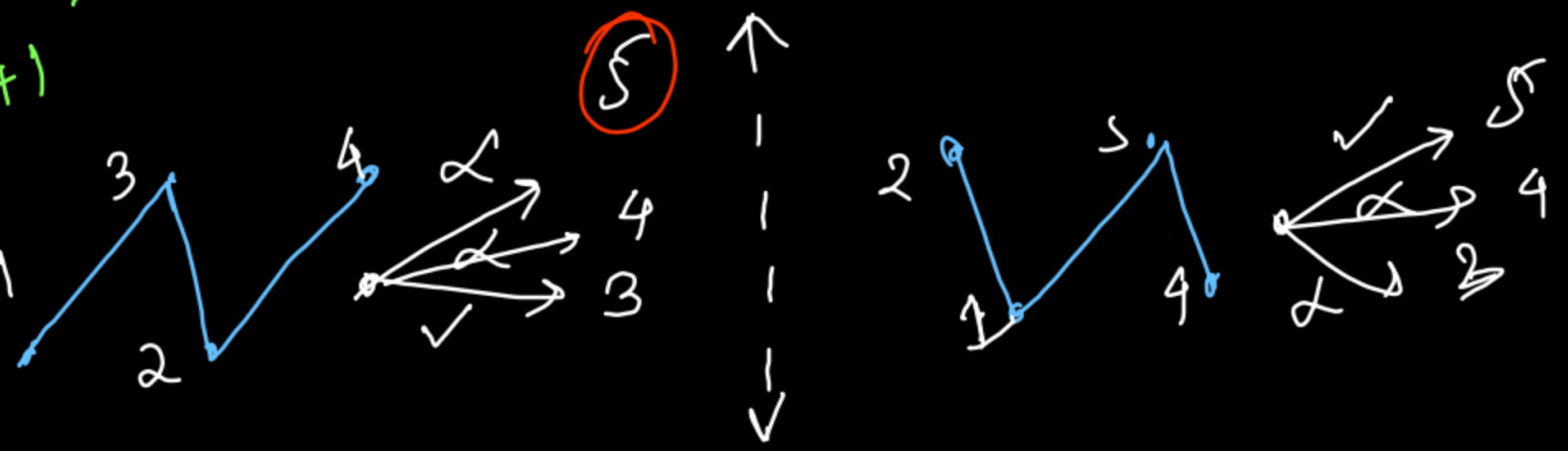
I → {1,5}
D → {1,17,5}

I → {1,17,5,13}
D → {1,17,13}

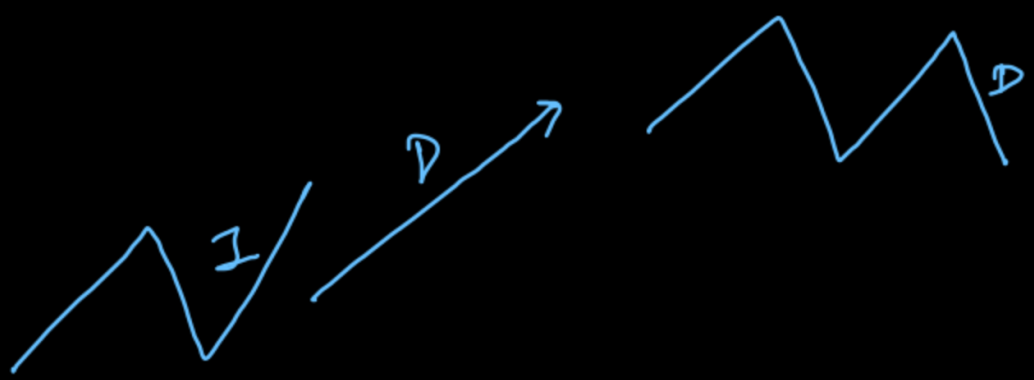
I → {1,17,5,10}
D → {1,17,5,13,10}

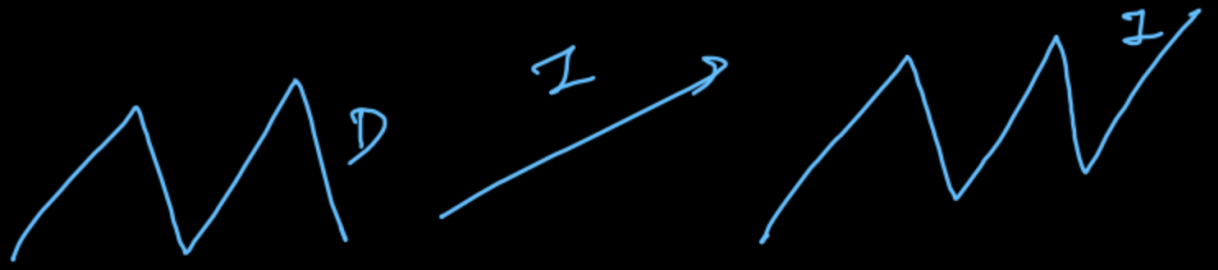
inc[i] = dec[j] + 1

dec[i] = inc[j] + 1



$dec[i] = inc[j] + 1$





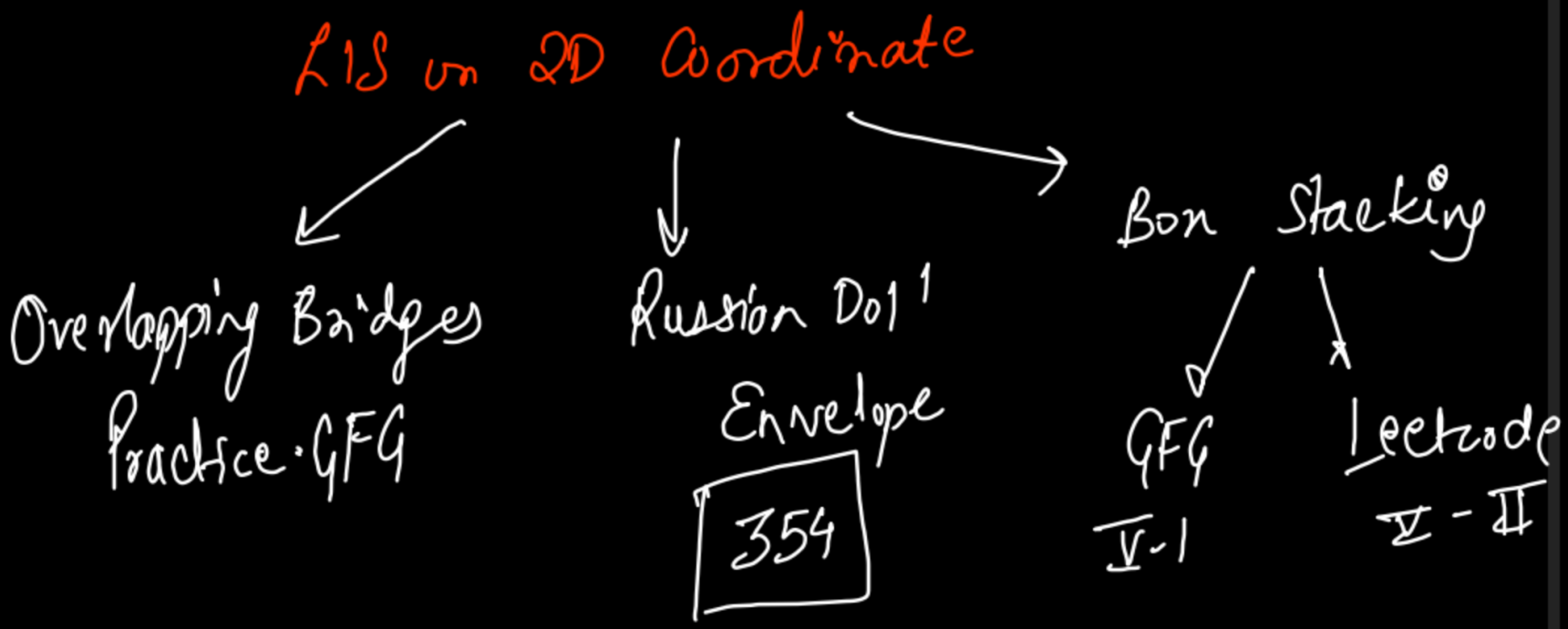
$inc[i] = dec[j] + 1$

```java
public int wiggleMaxLength(int[] nums) {
    int[] inc = new int[nums.length];
    Arrays.fill(inc, 1);

    int[] dec = new int[nums.length];
    Arrays.fill(dec, 1);

    int maxLength = 0;
    for(int i=0; i<nums.length; i++){

        for(int j=0; j<i; j++){
            if(nums[i] > nums[j]){
                // We are Increasing
                inc[i] = Math.max(inc[i], dec[j] + 1);
            } else if(nums[i] < nums[j]){
                // We are Decreasing
                dec[i] = Math.max(dec[i], inc[j] + 1);
            }
        }

        maxLength = Math.max(maxLength, Math.max(inc[i], dec[i]));
    }

    return maxLength;
}
```

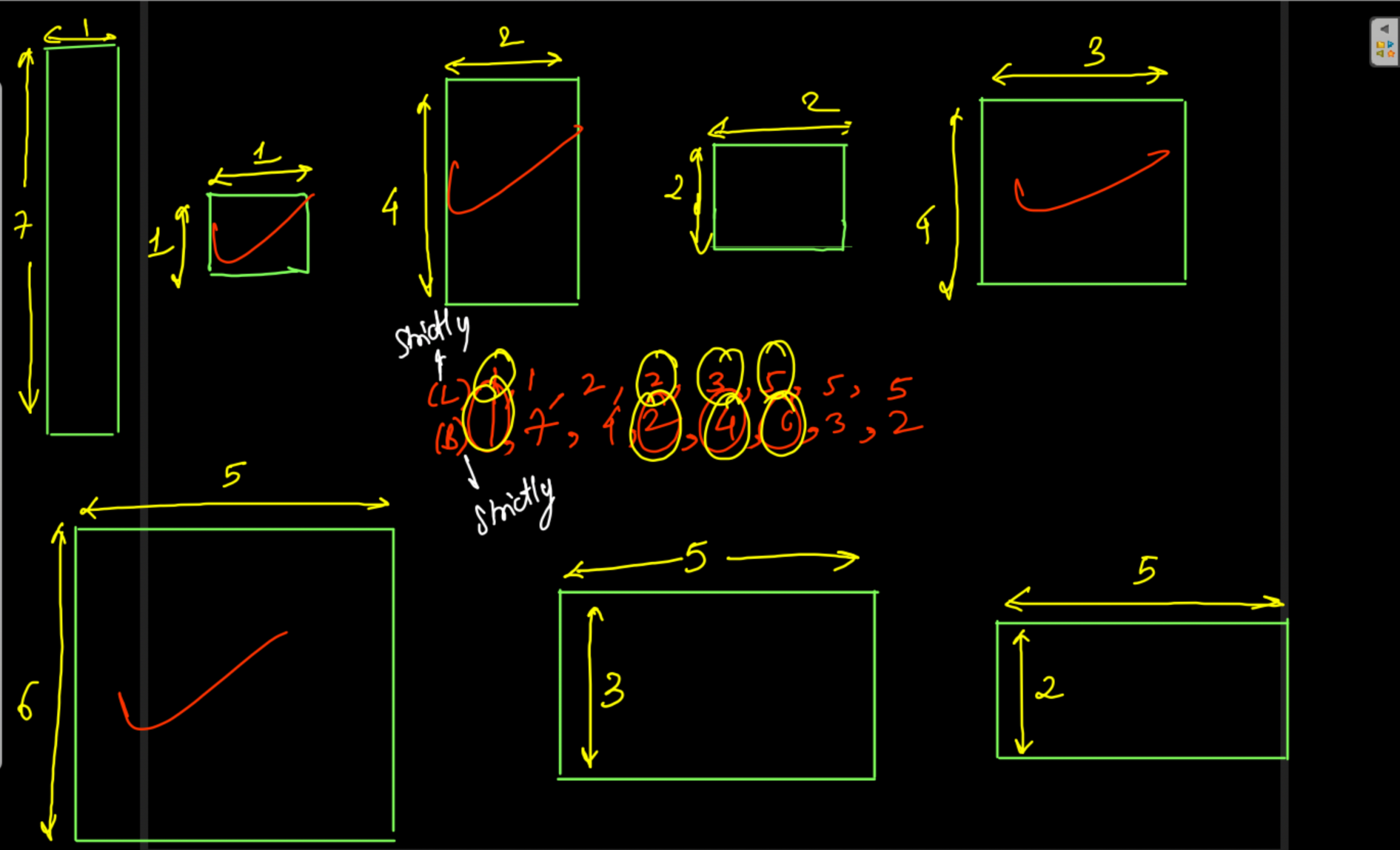
Time → $O(2*N^2)$
$= O(N^2)$

Space → $O(2*N)$

2 rows of 1D DP



One envelope can fit into another if and only if both the width and height of one envelope are greater than the other envelope's width and height.

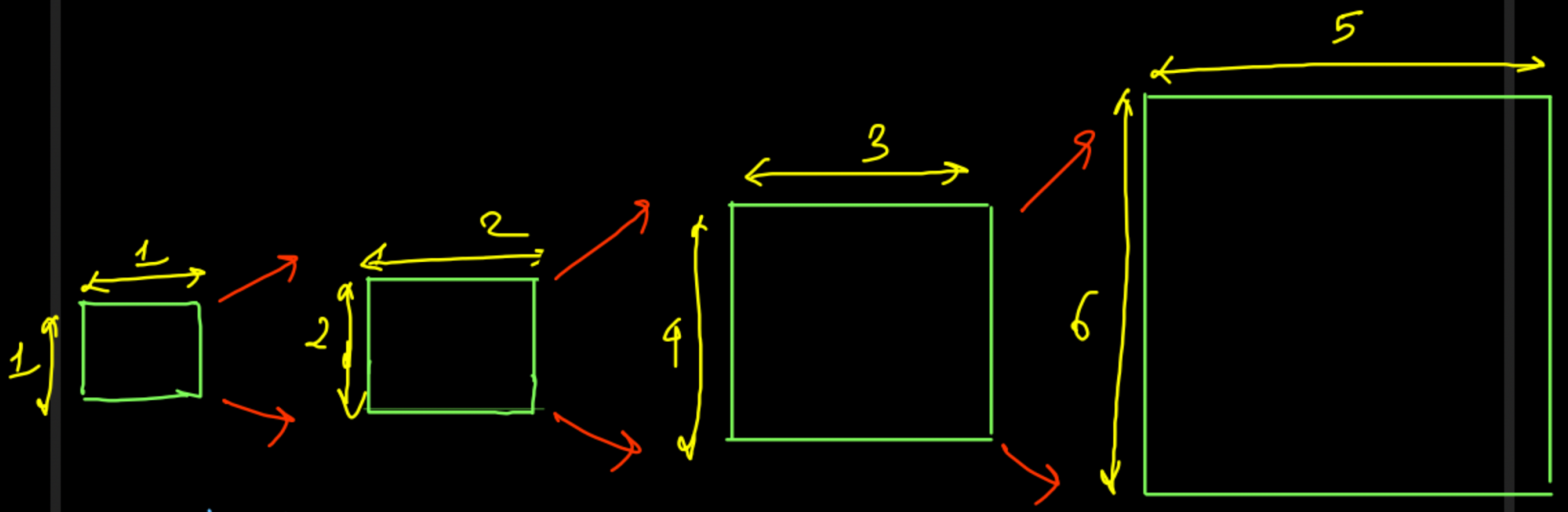

1
7
1
1
2
4
2
2
3
4
Strictly
(L) 1, 1, 2, 2, 3, 5, 5, 5
(B) 1, 7, 4, 2, 4, 6, 3, 2
Strictly
5
6
5
3
5
2

Time - O(NlgN + NlogN)
↓ Breadth ↓ length

Space → O(N)

```java
public int lowerBound(ArrayList<Integer> nums, int target){
    int low = 0, high = nums.size() - 1;
    int idx = nums.size();

    while(low <= high){
        int mid = low + (high - low) / 2;

        if(nums.get(mid) < target){
            low = mid + 1;
        } else {
            high = mid - 1;
            idx = mid;
        }
    }

    return idx;
}
```

```java
public int maxEnvelopes(int[][] envelops) {
    Arrays.sort(envelops, (int[] first, int[] second) -> {
        return (first[0] != second[0]) ? first[0] - second[0]
            : second[1] - first[1];
    });

    int n = envelops.length;
    ArrayList<Integer> sorted = new ArrayList<>();

    for(int i=0; i<envelops.length; i++){
        int lb = lowerBound(sorted, envelops[i][1]);
        if(lb == sorted.size()){
            sorted.add(envelops[i][1]);
            // Current Element larger than the largest
            // LIS of one length more
        } else {
            sorted.set(lb, envelops[i][1]);
        }
    }

    return sorted.size(); // This Sorted Array has same size as LIS
}
```



LIS on length

LIS on breadth

1, 2, 3, 5

1, 2, 4, 6

2D LIS

↳ 1st coordinate → Sorting

↳ 2nd coordinate → LIS

Longest Common Subsequence

Lecture - 1 Saturday (28 May)

9 AM to 12 PM

→ Length of LCS

→ Print LCS → Anyone
→ All

→ Longest Repeated Subset (LRS)

→ Palindromic Subsequence → Length
→ Count

Longest Common Subsequence

↳ order of characters

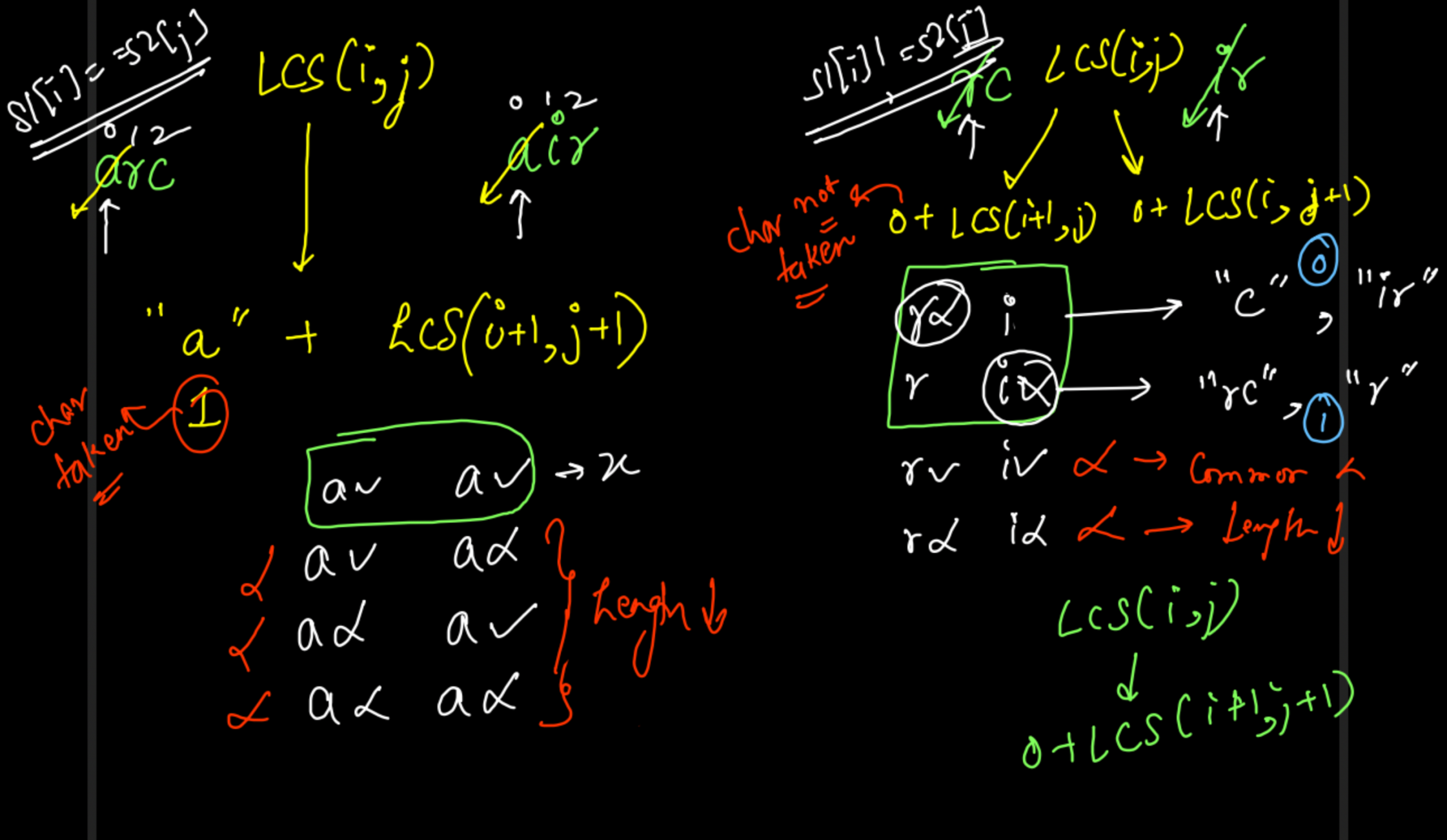
eg "abcde" , "bdca"

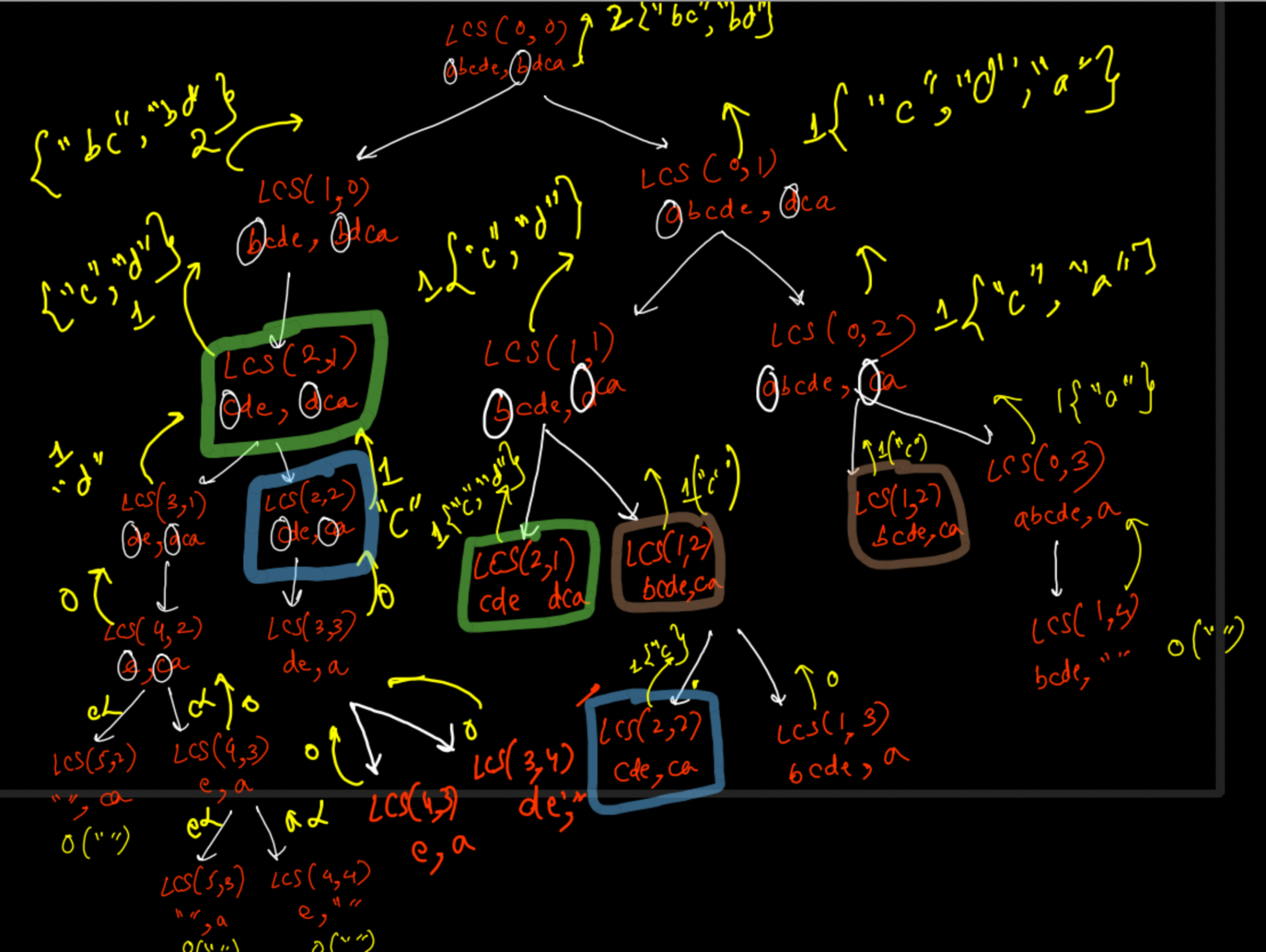
Brute force

$O(2^N * 2^M)$

↑

Compare all subsequences


s1[i] = s2[j]
LCS(i,j)
arc
air
"a" + LCS(i+1,j+1)
char taken = 1
ar ar → x
ar ac
ac ar
ac ac
length ↓
s1[i] != s2[j]
rc
LCS(i,j)
char not taken
0 + LCS(i+1,j)
0 + LCS(i,j+1)
"c" 0
"ir"
"rc" 1
"r"
rr ir → Common ↑
rc ic → Length ↓
LCS(i,j)
0 + LCS(i+1,j+1)

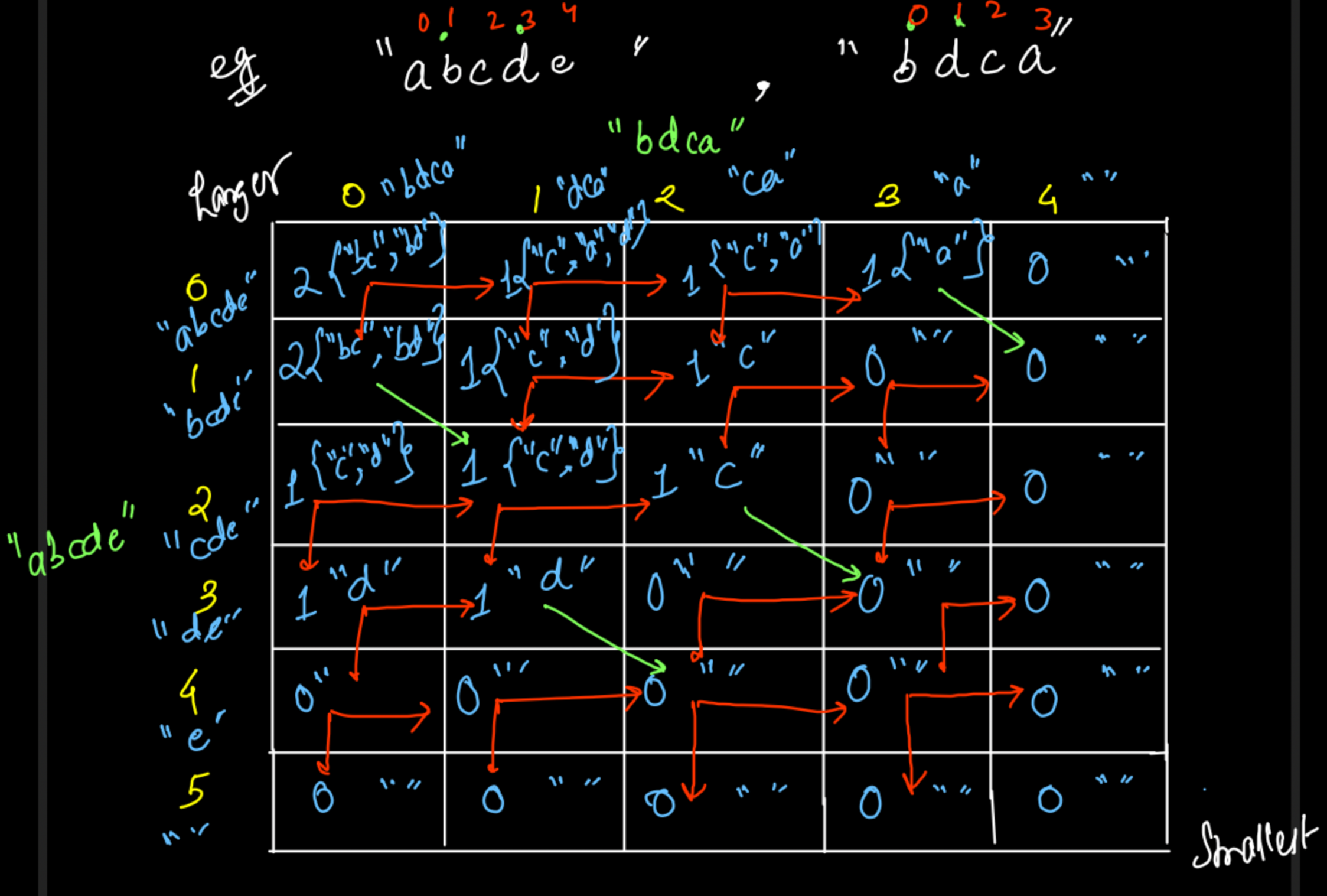

LCS(0,0)
abcde, bdca
2 {"bc","bd"}
{"bc","bd"}
2
LCS(1,0)
bcde, bdca
LCS(0,1)
abcde, dca
1{"c","d"}
1{"c","d"}
1
LCS(2,1)
cde, dca
LCS(1,1)
bcde, dca
LCS(0,2)
abcde, ca
1{"c","a"}
1{"c"}
LCS(3,1)
de, dca
LCS(2,2)
cde, ca
LCS(2,1)
cde dca
LCS(1,2)
bcde,ca
LCS(1,2)
bcde,ca
LCS(0,3)
abcde, a
1{"a"}
LCS(4,2)
e, ca
LCS(3,3)
de, a
LCS(3,4)
de, "
LCS(2,3)
cde, ca
LCS(1,3)
bcde, a
LCS(1,4)
bcde, ""
LCS(5,2)
"", ca
LCS(4,3)
e, a
LCS(4,3)
e, a
LCS(5,3)
"", a
LCS(4,4)
e, ""

```java
public int LCS(int i, int j, String s1, String s2, int[][] dp){
    if(i == s1.length() || j == s2.length())
        return 0; // LCS of Empty String with Other String is Empty String only

    if(dp[i][j] != -1) return dp[i][j];

    char ch1 = s1.charAt(i);
    char ch2 = s2.charAt(j);

    if(ch1 == ch2) // If characters are same, take the common from both of them
        return dp[i][j] = 1 + LCS(i + 1, j + 1, s1, s2, dp);

    // If character is uncommon, either not take s1[i] or not take s2[j]
    int option1 = LCS(i + 1, j, s1, s2, dp);
    int option2 = LCS(i, j + 1, s1, s2, dp);
    return dp[i][j] =  0 + Math.max(option1, option2);
}
```

```java
public int longestCommonSubsequence(String s1, String s2) {
    int[][] dp = new int[s1.length() + 1][s2.length() + 1];
    for(int i=0; i<dp.length; i++){
        for(int j=0; j<dp[0].length; j++){
            dp[i][j] = -1;
        }
    }

    return LCS(0, 0, s1, s2, dp);
}
```

Time → O(N*M)

Space → 2D DP
O(N*M)


eg "abcde" , "bdca"
0 1 2 3 4
0 1 2 3
longer
0 "bdca"
1 "dca"
2 "ca"
3 "a"
4 ""
"abcde"
"bdca"
"cde"
"de"
"e"
""
"a3cde"
Smallest

```java
public int longestCommonSubsequence(String s1, String s2) {
    int[][] dp = new int[s1.length() + 1][s2.length() + 1];

    for(int i=s1.length()-1; i>=0; i--){
        for(int j=s2.length()-1; j>=0; j--){
            char ch1 = s1.charAt(i);
            char ch2 = s2.charAt(j);

            if(ch1 == ch2)
                dp[i][j] = 1 + dp[i + 1][j + 1];

            else dp[i][j] = Math.max(dp[i + 1][j], dp[i][j + 1]);
        }
    }

    return dp[0][0];
}
```

Time → O(N*M)

Space → O(N*M)

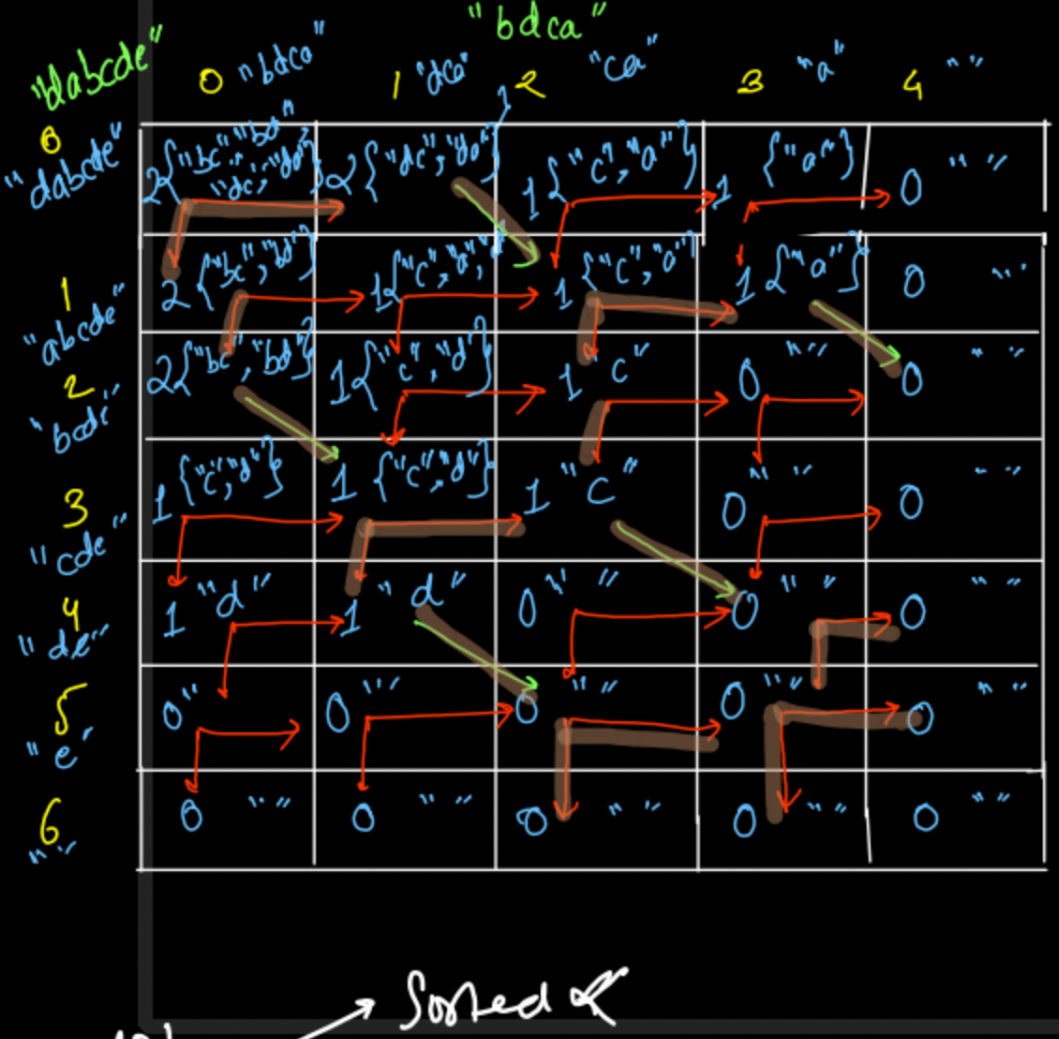
2D DP

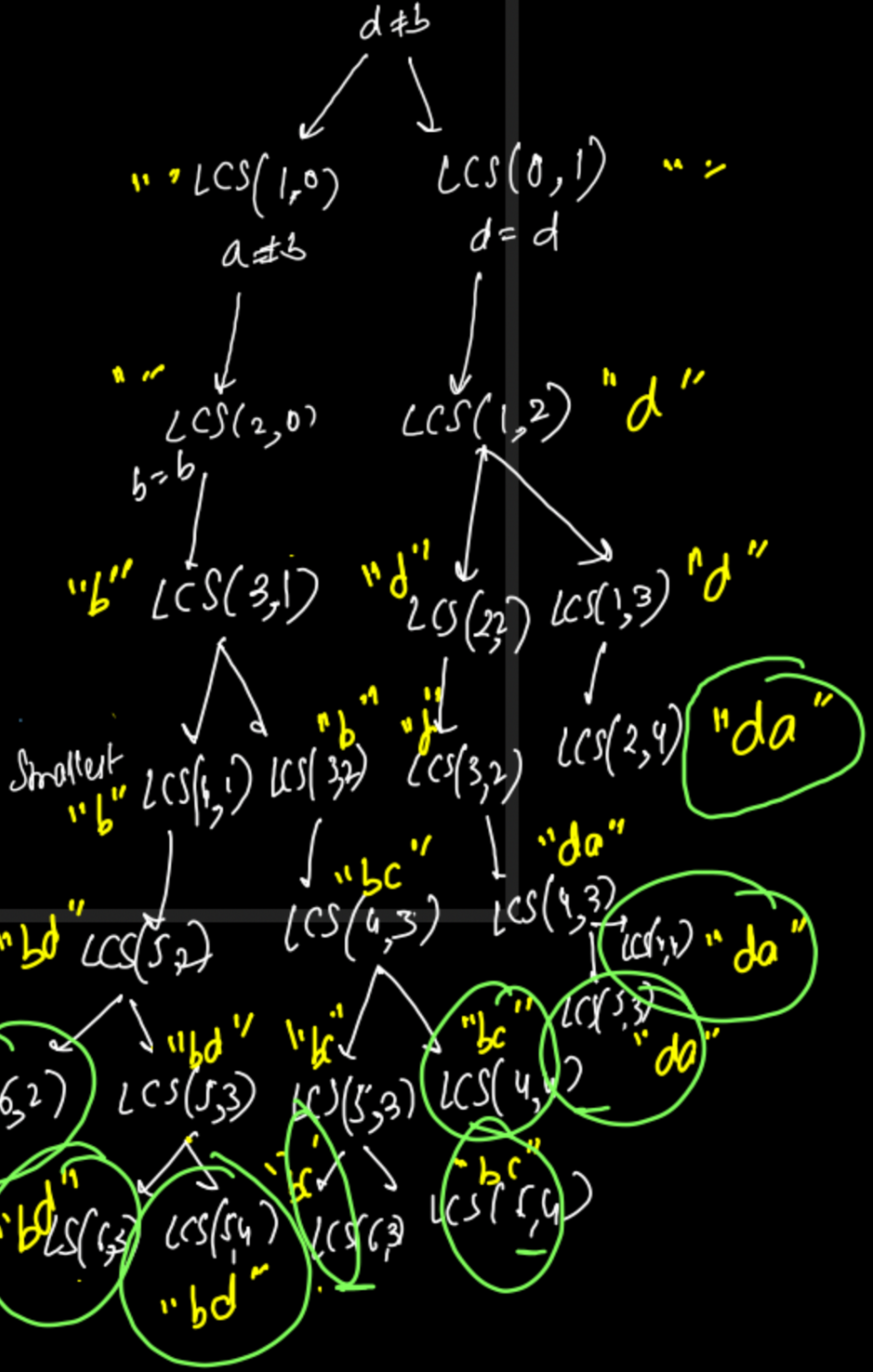
```java
int[] next = new int[s2.length() + 1];

for(int i=s1.length()-1; i>=0; i--){
    int[] curr = new int[s2.length() + 1];

    for(int j=s2.length()-1; j>=0; j--){
        char ch1 = s1.charAt(i);
        char ch2 = s2.charAt(j);

        if(ch1 == ch2)
            curr[j] = 1 + next[j + 1];

        else curr[j] = Math.max(next[j], curr[j + 1]);
    }

    next = curr;
}

return next[0];
```

Time → O(N*M)

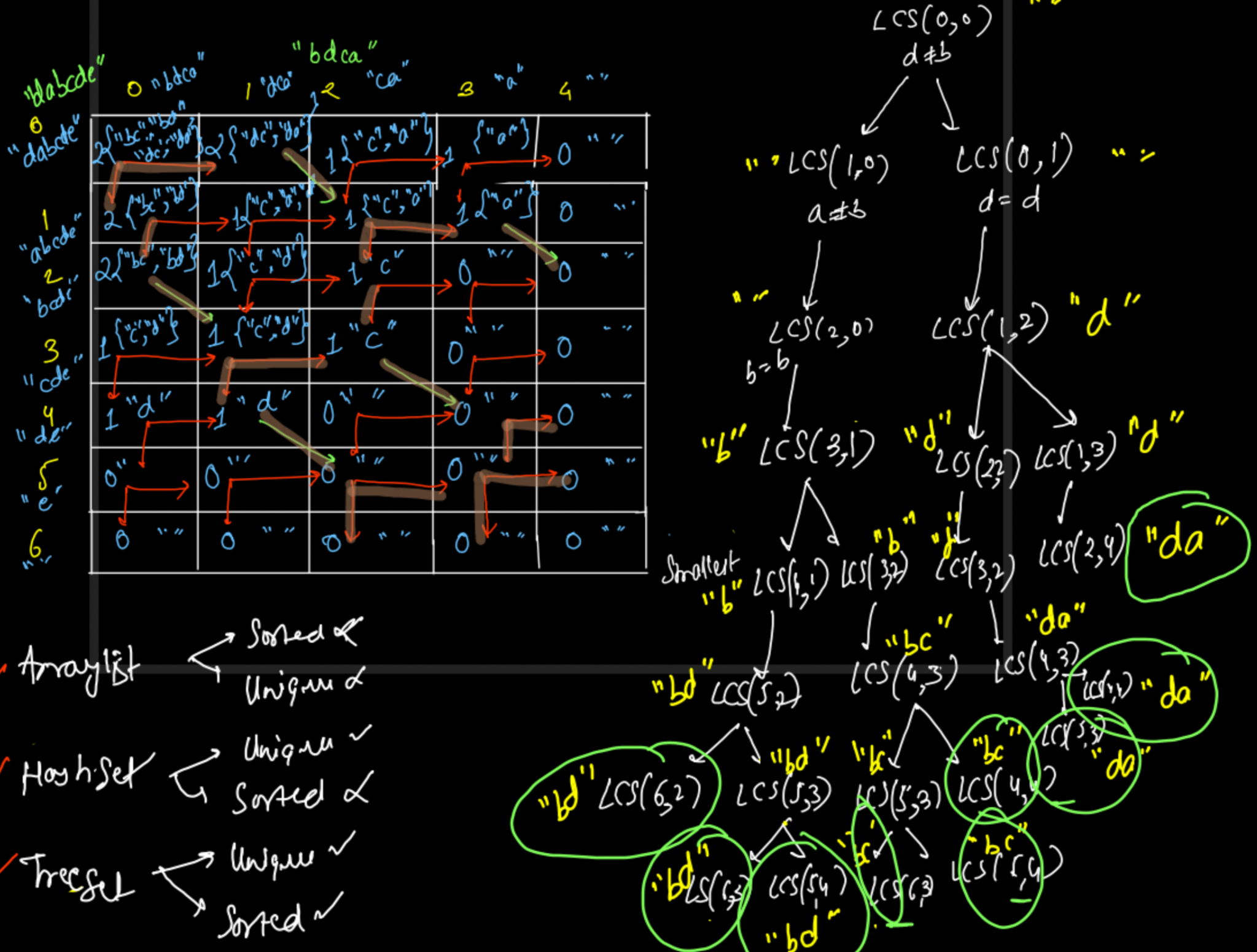
Space → O(2*M)

≈ 1D DP

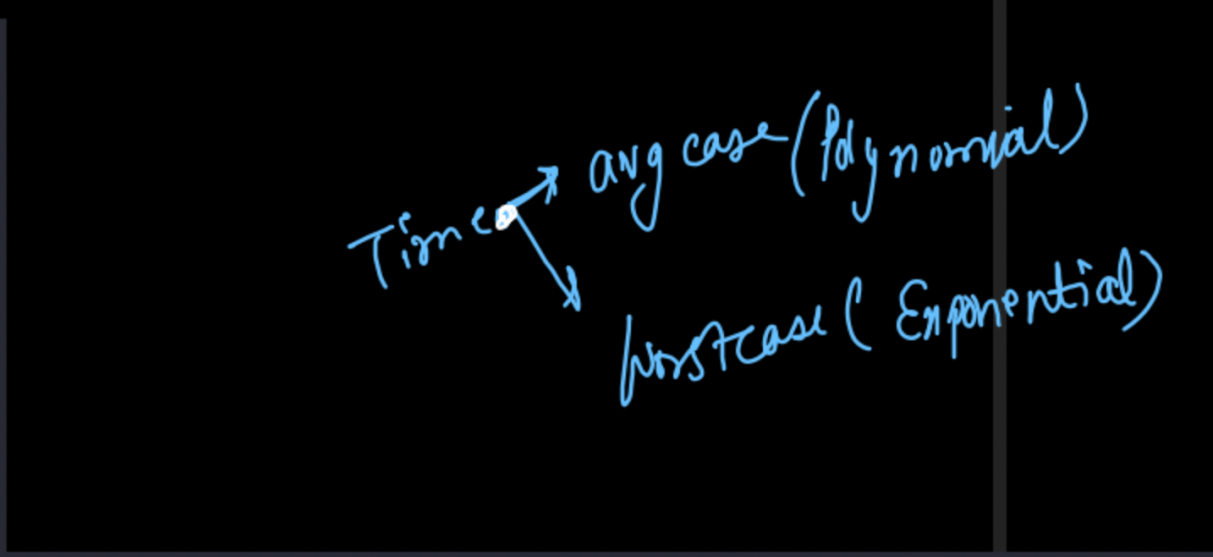




- ✓ ArrayList → Sorted ✗, Unique ✗
- ✓ HashSet → Unique ✓, Sorted ✗
- ✓ TreeSet → Unique ✓, Sorted ✓


"dabcde"
"bdca"
0 "bdca"
1 "dca"
2 "ca"
3 "a"
4 ""
6 "dabcde"
1 "abcde"
2 "bde"
3 "cde"
4 "de"
5 "e"
6 ""
LCS(0,0)
d≠b
LCS(1,0)
LCS(0,1)
a≠b
d=d
LCS(2,0)
LCS(1,2)
b=b
"b"
LCS(3,1)
"d"
LCS(2,2)
LCS(1,3)
"d"
Smallest
"b"
LCS(4,1)
LCS(3,2)
LCS(2,4)
"da"
"bd"
LCS(5,2)
"bc"
LCS(4,3)
"da"
LCS(3,4)
"da"
"bd"
LCS(6,2)
"bd"
LCS(5,3)
"b"
LCS(5,3)
"bc"
LCS(4,4)
"bd"
LCS(5,4)
"bc"
LCS(6,3)
"bc"
LCS(5,4)
"bd"
ArrayList
Sorted ✗
Unique ✗
HashSet
Unique ✓
Sorted ✗
TreeSet
Unique ✓
Sorted ✓

```java
TreeSet<String> answers; // Both Ordered (Lexicographical Order), Unique

public void helper(int i, int j, String s1, String s2, int[][] dp, String lcs){
    if(i == s1.length() || j == s2.length()){
        answers.add(lcs);
        return;
    }

    char ch1 = s1.charAt(i);
    char ch2 = s2.charAt(j);

    if(ch1 == ch2){
        // Character Taken (Same)
        helper(i + 1, j + 1, s1, s2, dp, lcs + ch1);
    } else {
        // Character Not Taken
        if(dp[i][j] == dp[i + 1][j]){
            helper(i + 1, j, s1, s2, dp, lcs);
        }

        if(dp[i][j] == dp[i][j + 1]){
            helper(i, j + 1, s1, s2, dp, lcs);
        }
    }
}
```




```java
public List<String> all_longest_common_subsequences(String s1, String s2)
{
    int[][] dp = new int[s1.length() + 1][s2.length() + 1];

    for(int i=s1.length()-1; i>=0; i--){
        for(int j=s2.length()-1; j>=0; j--){
            char ch1 = s1.charAt(i);
            char ch2 = s2.charAt(j);

            if(ch1 == ch2)
                dp[i][j] = 1 + dp[i + 1][j + 1];

            else dp[i][j] = Math.max(dp[i + 1][j], dp[i][j + 1]);
        }
    }

    answers = new TreeSet<>();
    helper(0, 0, s1, s2, dp, "");
    List<String> res = new ArrayList<>();
    for(String str: answers){
        res.add(str);
    }
    return res;
```

718 Longest Common Substring

Contiguous

eg1

"dabcde" vs "bdca"

Substring Subsequence

bc → ✓ ✓

bd → ✗ ✓

dc → ✗ ✓

da → ✓ ✗

Longest common substring → "d", "a", "b", "c"

eg2

"abcdabc" , "bcdbca"

abc , bca

bc

"abcd" "abc" , "bcd" "bca"

s1[i] == s2[j]

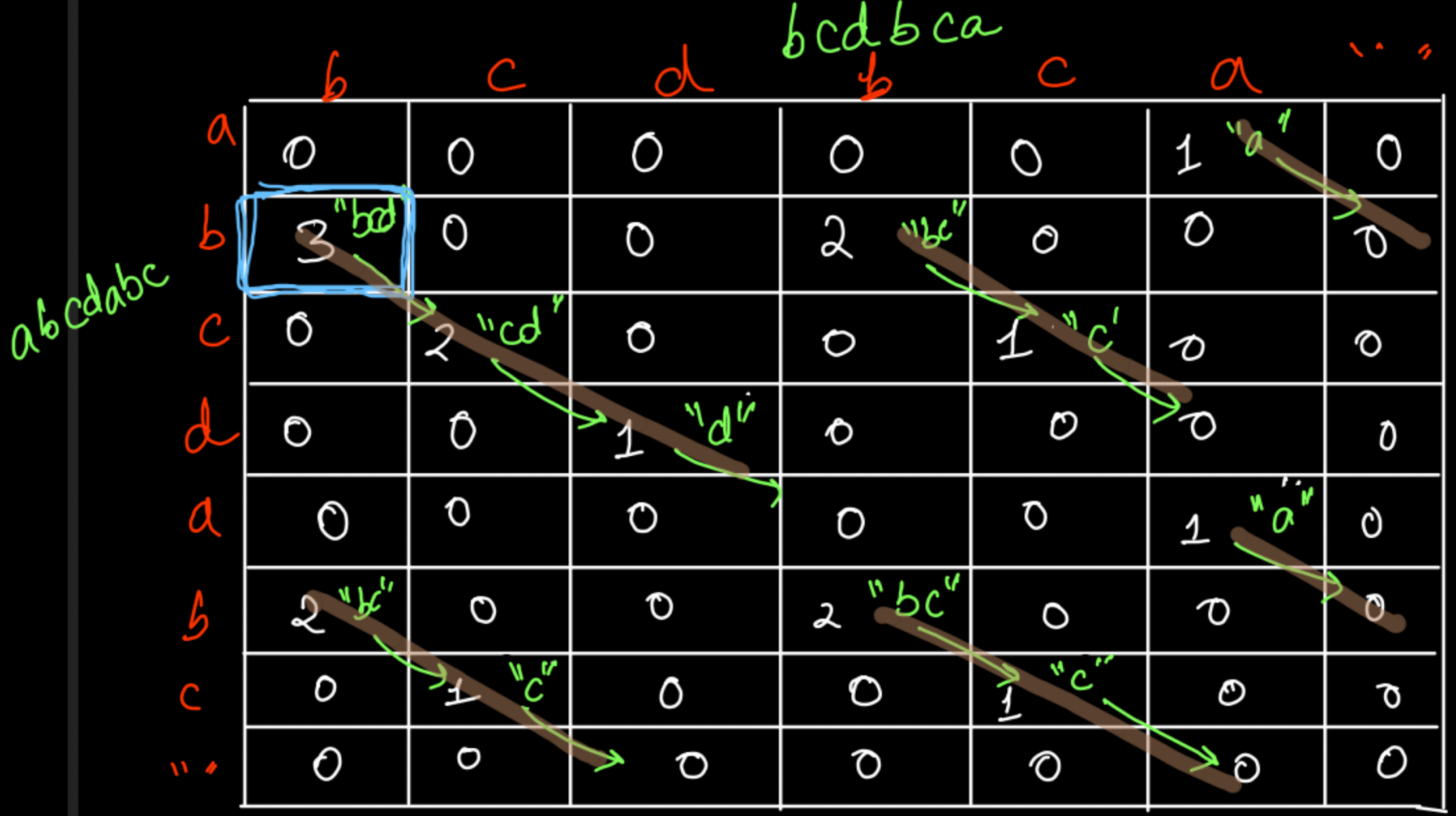
s1[i] != s2[j]

LCSS(i,j)

1 + LCSS(i+1, j+1)

LCSS(i+1, j)

LCSS(i, j+1)


bcdbca
b c d b c a ''
abcdabc
a
b
c
d
a
b
c
''
"bcd"
"cd"
"d"
"bc"
"c"
"a"
"bc"
"c"
"a"
"bc"
"c"

```java
class Solution {
    public int findLength(int[] s1, int[] s2) {
        int[][] dp = new int[s1.length + 1][s2.length + 1];

        int maxLen = 0;
        for(int i=s1.length-1; i>=0; i--){
            for(int j=s2.length-1; j>=0; j--){
                if(s1[i] == s2[j])
                    dp[i][j] = 1 + dp[i + 1][j + 1];

                // else dp[i][j] = Math.max(dp[i + 1][j], dp[i][j + 1]);
                maxLen = Math.max(maxLen, dp[i][j]);
            }
        }

        return maxLen;
    }
}
```

Time → O(N*M)

Space → O(N*M)

2D DP

⭐ It can be space optimized to 1D DP

Longest Common Subsequence - II

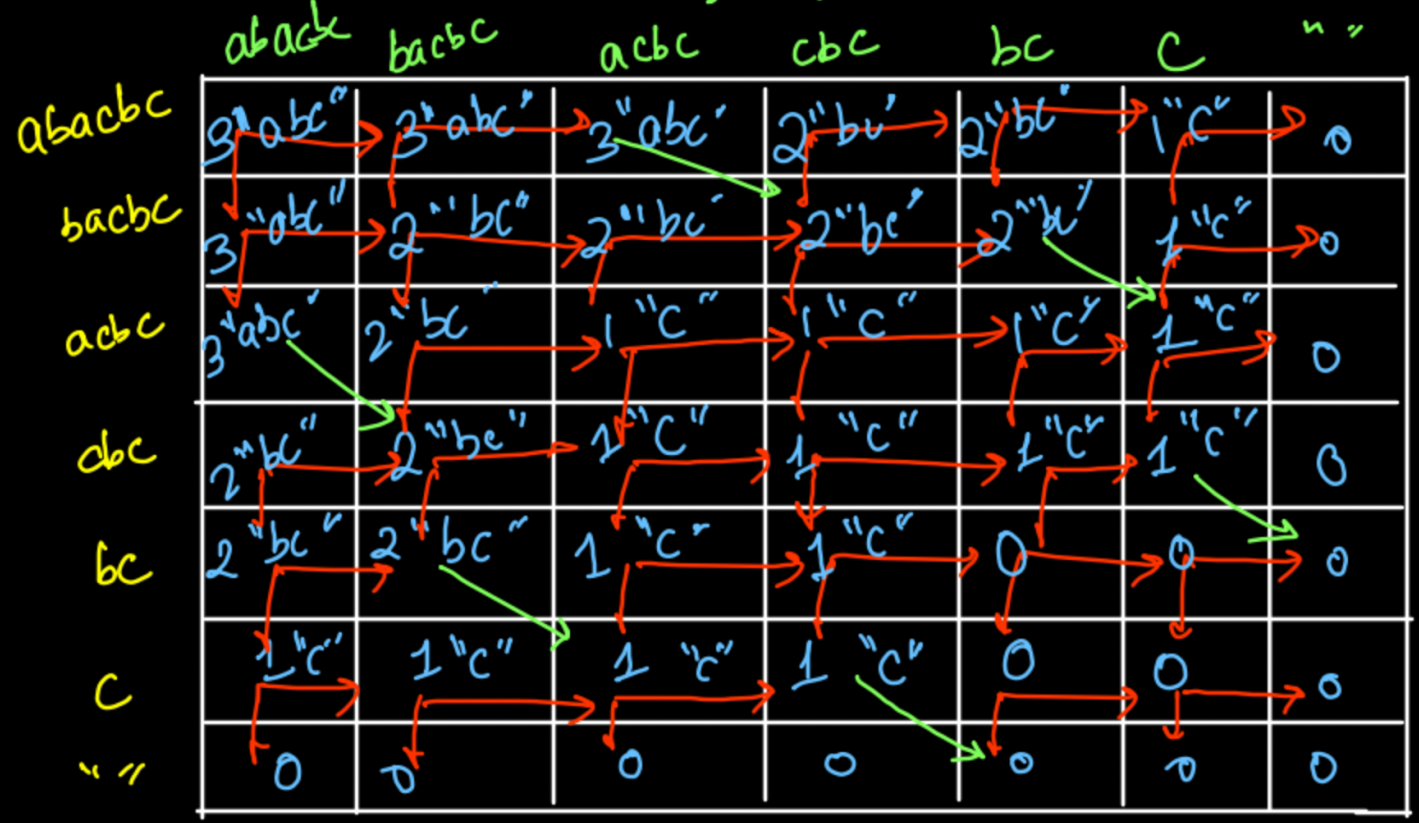
29th may, Sunday, 9 AM - 12 PM

- Longest Repeating / Duplicate Subsequence (GFG)
- longest Duplicate String
  - I (1044 LC)
  - II (GFG)
- longest Palindromic Subsequence (LC 516)
- min^m Steps to Make Palindrome
  - Insert^n (GFG)
  - Delete^n (1312 LC)
- Palindromic Substrings
  - longest (5 LC)
  - Count (647 LC)

Longest Repeating Subsequence

"abacbc"

LCS



diff char
dp[i][j] = dp[i+1][j+1]

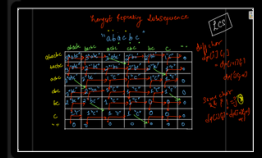
dp[i][j+1]

same char
&& i != j

dp[i][j] = dp[i+1][j+1] + 1

```java
public int LongestRepeatingSubsequence(String s)
{
    int[][] dp = new int[s.length() + 1][s.length() + 1];

    for(int i=s.length()-1; i>=0; i--){
        for(int j=s.length()-1; j>=0; j--){
            char ch1 = s.charAt(i);
            char ch2 = s.charAt(j);

            if(ch1 == ch2 && i != j)
                dp[i][j] = 1 + dp[i + 1][j + 1];

            else dp[i][j] = Math.max(dp[i + 1][j], dp[i][j + 1]);
        }
    }

    return dp[0][0];
}
```

longest Repeating subset
↳ LCS with same string



Longest Repeating Substring

Leetcode 1044

→ Non-overlapping (GFG)

→ Overlapping (Leetcode 1044)

eg: "banana"

		b	a'	n'	a''	n''	a'''	''
		0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0
a'	0	0	0	3 "ana"	0	1 "a"	0	
n'	0	0	0	0	2 "na"	0	0	
a''	0	3 "ana"	0	0	0	1 "a"	0	
n''	0	0	2 "na"	0	0	0	0	
a'''	0	1 "a"	0	1 "a"	0	0	0	
''	0	0	0	0	0	0	0	0

Longest common substring on 2 same strings

Meaning: dp[i][j] = LCS of string s1.substring(i) & s2.substring(j)

0.7 x

1044

```java
public String longestDupSubstring(String s) {
    int[][] dp = new int[s.length() + 1][s.length() + 1];

    int idx = s.length(), len = 0;

    for(int i=s.length()-1; i>=0; i--){
        for(int j=s.length()-1; j>=0; j--){
            char ch1 = s.charAt(i);
            char ch2 = s.charAt(j);

            if(ch1 == ch2 && i != j)
                dp[i][j] = 1 + dp[i + 1][j + 1];

            if(dp[i][j] > len){
                idx = i;
                len = dp[i][j];
            }

            // else dp[i][j] = Math.max(dp[i + 1][j], dp[i][j + 1]);
        }
    }

    return s.substring(idx, idx + len);
}
```

(TLE) → Expected Rolling Hash

Time → $O(N^2)$

Space → $O(N^2)$ 2D DP

Longest Repeated / Duplicate Substrings with non-overlapping (GFG)

"banana" Length = 2 "an"

	b	a'	n'	a''	n''	a'''	''''
b	0	0	0	0	0	0	0
a'	0	0	0	3 "ana"	0	1 "a"	0
n'	0	0	0	0	2 "na"	0	0
a''	0	3 "ana"	0	0	0	1 "a"	0
n''	0	0	2 "na"	0	0	0	0
a'''	0	1 "a"	0	1 "a"	0	0	0
''''	0	0	0	0	0	0	0

max(i,j) + dp[i][j]

max(i,j)

```java
int[][] dp = new int[s.length() + 1][s.length() + 1];
int idx = s.length(), len = 0;

for(int i=s.length()-1; i>=0; i--){
    for(int j=s.length()-1; j>=0; j--){
        char ch1 = s.charAt(i);
        char ch2 = s.charAt(j);

        if(ch1 == ch2 && i != j)
            dp[i][j] = 1 + dp[i + 1][j + 1];

        if(Math.min(i, j) + dp[i][j] >= Math.max(i, j)){
            // Overlapping Substrings
            dp[i][j] = 0;
        }

        if(dp[i][j] >= len){
            idx = i;
            len = dp[i][j];
        }
    }
}
```

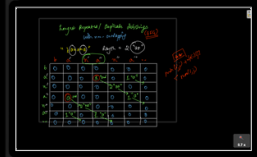
```java
if(len == 0) return "-1";
return s.substring(idx, idx + len);
```

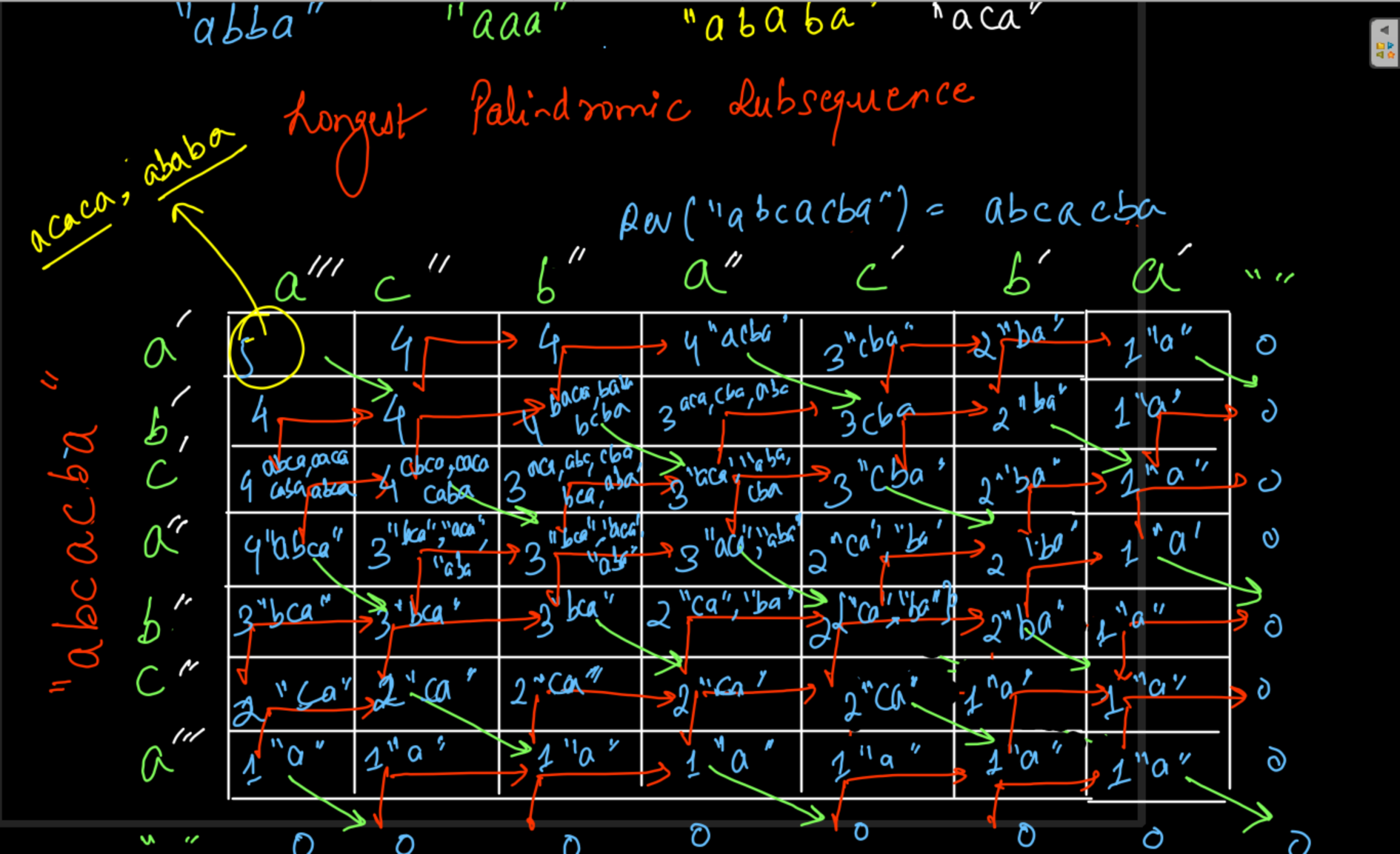
abs(j-i) ≤ dp(i)(j)

To compute only non-overlapping substrings

Time → O(n²)

Space → O(n²)



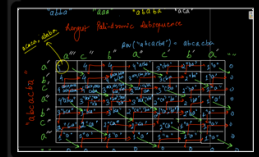

"abba"
"aaa"
"ababa"
"aca"
Longest Palindromic Subsequence
acaca, ababa
Rev("abcacba") = abcacba
"abcacba"
a''' c'' b'' a'' c' b' a'
a' b' c' a'' b'' c'' a'''
5
4 4 4 "acba" 3 "cba" 2 "ba" 1 "a"
4 4 4 bacaa, baba, bcba 3 aca, cba, abc 3 cba 2 "ba" 1 "a"
4 abca, acca, caba, aba 4 abco, acco, caba 3 aca, abc, cba, bca, aba 3 "aca", "aba", cba 3 "cba" 2 "ba" 1 "a"
4 "abca" 3 "bca", "aca", "aba" 3 "bca", "bca", "aba" 3 "aca", "aba" 2 "ca", "ba" 2 "ba" 1 "a"
3 "bca" 3 "bca" 3 "bca" 2 "ca", "ba" 2 "ca", "ba" 2 "ba" 1 "a"
2 "ca" 2 "ca" 2 "ca" 2 "ca" 2 "ca" 1 "a" 1 "a"
1 "a" 1 "a" 1 "a" 1 "a" 1 "a" 1 "a" 1 "a"
0 0 0 0 0 0 0

```java
public int LCS(String s1, String s2){
    int[][] dp = new int[s1.length() + 1][s2.length() + 1];

    for(int i=s1.length()-1; i>=0; i--){
        for(int j=s2.length()-1; j>=0; j--){
            char ch1 = s1.charAt(i);
            char ch2 = s2.charAt(j);

            if(ch1 == ch2)
                dp[i][j] = 1 + dp[i + 1][j + 1];

            else dp[i][j] = Math.max(dp[i + 1][j], dp[i][j + 1]);
        }
    }

    return dp[0][0];
}
public int longestPalindromeSubseq(String s) {
    StringBuilder rev = new StringBuilder(s);
    rev = rev.reverse();
    String s2 = new String(rev);

    return LCS(s, s2);
}
```

LPS
= LCS(S, rev(S))

Time → O(N²)

Space → O(N²)



"a b c a b c a"

Min Deletions to make Palindrome

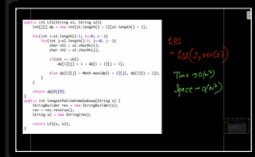
Min Insertions to make Palindrome

"a b c a b c a"

Ans = Str.length - LPS

"a b c b a b c b a"

Ans = Str.length - LPS



```java
class Solution {
    public int LCS(String s1, String s2){
        int[][] dp = new int[s1.length() + 1][s2.length() + 1];

        for(int i=s1.length()-1; i>=0; i--){
            for(int j=s2.length()-1; j>=0; j--){
                char ch1 = s1.charAt(i);
                char ch2 = s2.charAt(j);

                if(ch1 == ch2)
                    dp[i][j] = 1 + dp[i + 1][j + 1];

                else dp[i][j] = Math.max(dp[i + 1][j], dp[i][j + 1]);
            }
        }

        return dp[0][0];
    }

    public int minInsertions(String s) {
        StringBuilder rev = new StringBuilder(s);
        rev = rev.reverse();
        String s2 = new String(rev);

        return s.length() - LCS(s, s2);
    }
}
```

Min insertions/ Deletions to make Palindrome

Longest Palindromic Substring

"abcabac"

LPSS = "aba" → 3

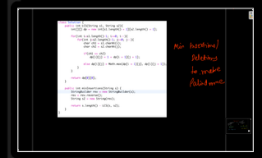
LPSS = LCSS(S, rev(S))

LPSS → Longest Palindromic Substring

LCSS → Longest Common Subsequence

Time → $O(N^2)$

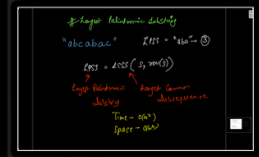
Space → $O(N^2)$



```java
public String longestCommonSubstring(String s1, String s2){
    int[][] dp = new int[s1.length() + 1][s2.length() + 1];

    String res = "";

    for(int i=s1.length()-1; i>=0; i--){
        for(int j=s2.length()-1; j>=0; j--){
            char ch1 = s1.charAt(i);
            char ch2 = s2.charAt(j);

            if(ch1 == ch2)
                dp[i][j] = 1 + dp[i + 1][j + 1];

            if(dp[i][j] >= res.length()){
                String curr = s1.substring(i, i + dp[i][j]);

                if(isPalindrome(curr) == true){
                    res = curr;
                }
            }
        }
    }

    return res;
}
```

```java
public String longestPalindrome(String s) {
    StringBuilder rev = new StringBuilder(s);
    rev = rev.reverse();
    String s2 = new String(rev);

    return longestCommonSubstring(s, s2);
}
```

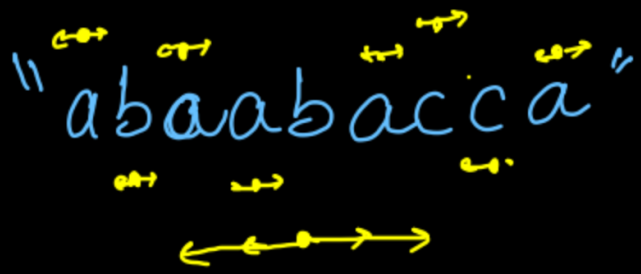
type of DP
This might go
wrong for
some testcases



Expand Around Center

Greedy

odd length palindromic substrings



↳ Treat each character as center for odd length palindromes

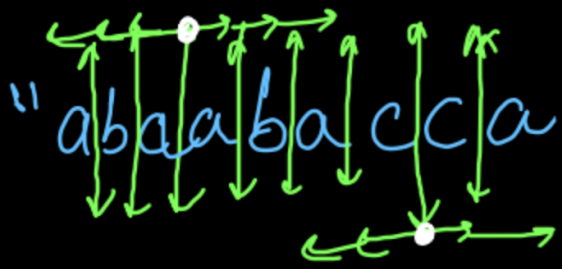
(odd)

LPSS = "a" / "b" / "c"

"aba"

"cabac"

even length palindromic substrings



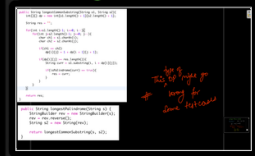
↳ Take in b/w 2 characters as middle for even length palindromes

(even)

LPSS = ""

"abba"

"abaaba"



Time → O(N²) → N * O(N) middle, Expand time in worst case, Space → O(1)

```
String res = "";
for(int i=0; i<s.length(); i++){
    // Odd Length Palindromes
    int left = i - 1, right = i + 1, len = 1;
    while(left >= 0 && right < s.length()){
        if(s.charAt(left) == s.charAt(right)){
            len = len + 2;
            left--; right++;
        } else {
            break;
        }
    }

    if(len > res.length()){
        res = s.substring(left + 1, left + 1 + len);
    }
}
```

```
for(int i=0; i<s.length(); i++){
    // Even Length Palindromes
    if(i + 1 == s.length() || s.charAt(i) != s.charAt(i + 1)){
        continue;
    }

    int left = i - 1, right = i + 2, len = 2;
    while(left >= 0 && right < s.length()){
        if(s.charAt(left) == s.charAt(right)){
            len = len + 2;
            left--; right++;
        } else break;
    }

    if(len > res.length()){
        res = s.substring(left + 1, left + 1 + len);
    }
}


return res;
```